View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

INCREMENTAL PROCESSING OF TWIG QUERIES

MANESH SUBHASH

(B.E. - Computer Science and Engineering, V.T.U. Karnataka, India)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgments

I thank my supervisor Prof. Chan Chee Yong for his continued support, encouragement and direction. I would also like to thank the professors who have taught me courses related to databases. It has indeed kept me motivated and focused on research related to databases.

I thank my dad Prof. Subhash Jacob, my mentor for all these years, for everything he has been to me. I would like to say a big thanks to my family and friends, whose continuous backing helps me to achieve my goals. This thesis would not have been possible without the omnipresent faith of my dear Sravanthy. Finally, God, whose blessing of good health has helped me complete this thesis at this hour.

Contents

Acknowledgments **ii**

Summary **ix**

1 Introduction **1**

1.1 Querying the XML database 1

1.2 Thesis Contributions 4

1.3 Thesis Organization 5

2 Related Work **6**

2.1 XML query processing using structural and holistic joins 7

2.2 Selectivity Estimation of Twig Queries 9

2.3 Incremental validation of XML schema 9

2.4 Discussion 10

3 Querying using pre-computations **12**

3.1 Preliminaries 12

3.2 The pre-computation model 13

3.3 Definitions and data structures 17

3.3.1 The probe:A pre-computation data structure 17

3.3.2 Representation of XML query 19

3.4 Overview of NodeMatch and PathMatch algorithms 20
3.5 The NodeMatch Algorithm 21
3.6 Incremental maintenance of NodeMatch 29
3.6.1 Insertion of a complete sub-tree using NodeMatch 30
3.6.2 Deletion of a complete sub-tree using NodeMatch 32
3.6.3 Complexity analysis of NodeMatch algorithm 35
3.7 The PathMatch algorithm 37
3.8 Incremental maintenance of PathMatch 38
3.8.1 Insertion of a complete sub-tree using PathMatch 39
3.8.2 Deletion of a complete sub-tree using PathMatch 40
3.8.3 Complexity analysis of PathMatch algorithm 41

4 Experimental study 43

4.1 Experimental setup 43
4.1.1 The data-sets 44
4.1.2 The boolean twig queries and update operations 45
4.2 Experiments and Results 47
4.2.1 Performance on various queries 48
4.2.2 Update Performance 50
4.2.3 Validation Time 51
4.2.4 Comparison of Space Requirements 54
4.2.5 Update times for varying Fan-out with constant Depth 55
4.2.6 Update times for varying depth with constant fan-out 56
4.2.7 Scalability Comparison 58
4.3 Summary 59

5 Conclusion 62

A Niagara XML Data Generator **64**

A.1 Configuration file template 64

List of Figures

1.1 Example of a XML document represented as a tree 2

1.2 Example of a reduced XML document 2

1.3 Example of a Twig query 3

3.1 Another example of a Twig query 13

3.2 Recursive procedure to check if a solution exists. 14

3.3 Example of node storing the maximal subtree match 14

3.4 Pre-computation of an XML document for query Q 15

3.5 The structure of a stored probe 18

3.6 Use of the two lists in the probe structure 18

3.7 NodeMatch and PathMatch storing probes 21

3.8 Function find_pattern() 24

3.9 Function create_probe() 25

3.10 Function prune_probe 25

3.11 Function set_next_position() 25

3.12 Function forward_to_next_level() 27

3.13 Function find_best_match_and_store() 27

3.14 Function check_for_extension() 28

3.15 Function compute_counts_and_merge() 28

3.16 Function insert_subtree() for NodeMatch 31

3.17 Function correct_parent_increment() for NodeMatch 31
3.18 Function delete_subtree() for NodeMatch 33
3.19 Function find_desc_matches() 34
3.20 Function correct_parent_decrement() 34
3.21 Function check_for_extension_new() 38
3.22 Function check_ancestor_exists() 39
3.23 Function delete_subtree() for PathMatch 40

4.1 Pre-computations for Data-set1 on Queries Q1-Q6 48
4.2 Pre-computations for Data-set2 on Queries Q1-Q6 48
4.3 Pre-computations for Data-set3 on Queries Q1-Q6 49
4.4 Pre-computations for Data-set1 on Queries Q7-Q15 49
4.5 Pre-computations for Data-set2 on Queries Q7-Q15 50
4.6 Pre-computations for Data-set3 on Queries Q7-Q15 50
4.7 Delete operations on Data-set1 51
4.8 Delete operations on Data-set2 52
4.9 Delete operations on Data-set3 52
4.10 Validation time for delete operations. 53
4.11 Insert operations on Data-set1 53
4.12 Insert operations on Data-set2 54
4.13 Insert operations on Data-set3 54
4.14 Memory requirements for increased repetition of element tags 55
4.15 Effect of varying the fan-out on delete operations 56
4.16 Effect of varying the fan-out on insert operations 56
4.17 Effect of varying the depth on delete operations 57
4.18 Effect of varying the depth on insert operations 57

4.19 Pre-computation on large data-sets 58

4.20 Delete operations on large data-sets 58

4.21 Insert operations on large data-sets 59

Summary

Queries on XML databases are typically expressed as a twig pattern. The XML database in itself can be modelled into a tree representation. The query processing problem then reduces to finding all occurrences of these twig patterns in this tree representation of the XML database. In this thesis, we develop two algorithms that use pre-computation techniques to answer *boolean twig queries* on XML databases. The goal here is to determine if a pattern exists in the database rather than retrieve all the matching data corresponding to the query. We extend the pre-computation algorithms to include support for update operations such as inserts and deletes of sub-trees on the XML database. We use the technique of incremental maintenance to support efficient and feasible updates of the pre-computations. The two algorithms differ in the degree of pre-computations stored. In the first algorithm, only those nodes that match any node of the query store the pre-computations. In the second algorithm, any node that lies in between nodes of a solution stores the pre-computations. This essential difference is critical to the performance of the updates. The pre-computations at intermediate nodes prevents the costly 'downward search' of the XML database. The proposed algorithms have been implemented and experimental results have been collected and analyzed using various data-sets and queries.

Chapter 1

Introduction

1.1 Querying the XML database

The eXtensible Markup Language (XML) [4] standardized by the W3C [6] has gained tremendous popularity as both an information representation format and as an information exchange medium. The need to store, process and maintain large volumes of XML data have resulted in the database community developing specialized solutions to meet these challenges. Early efforts saw the extensions of techniques in relational databases [19, 30, 26] and object oriented databases [22] being applied for the semi-structured XML data. The inherent semi-structured property have limited this extension leading to the development of database architectures such as Tamino[25], Timber [20] and Natix [18] that have re-created a different form of a database that is characterized by natural properties of a database system while tuned to the properties of XML.

The XML data is hierarchical in structure and can be logically modelled as a tree (assuming IDREFS [4] are ignored). The nodes represent the XML elements and the edges represent the relationships between the elements. The leaf nodes correspond to the values and attributes of its parent node. Figure 1.1 illustrates an example of a XML document modelled as XML tree.

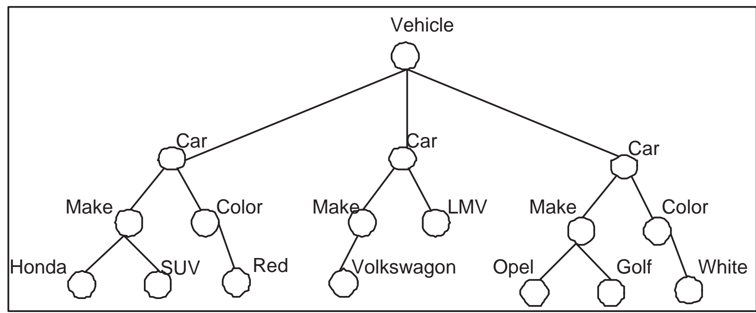

Figure 1.1: Example of a XML document represented as a tree

We can reduce this XML tree to contain only structural relationships. In this representation, each node in the tree contains in itself an element tag (the structural data) and its values and attributes (element data). For example, consider the element tag 'Make' shown in Figure 1.1. It has a value of 'Honda' and an attribute with value 'SUV'. The content and attribute values can be stored as part of the node matching the element tag. Using this representation the revised XML tree corresponding to Figure 1.1 is shown in Figure 1.2.

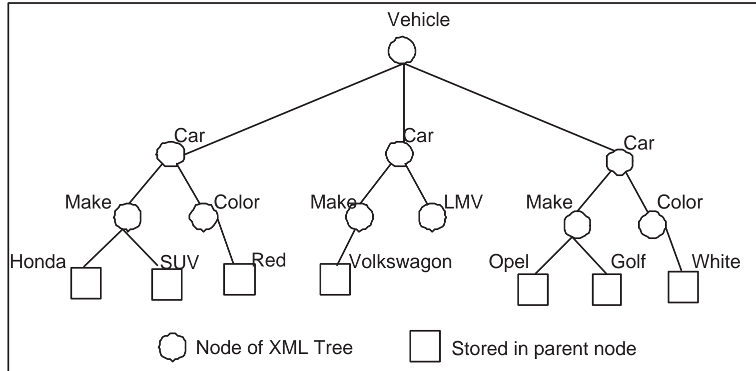

Figure 1.2: Example of a reduced XML document

Languages such as XPath [3] and XQuery [5] have been developed into standards that can be used to query data from the tree structured XML documents. These can be used for both structure and element data. Suppose we are given the XQuery expression

$$Car[cc = \text{“}2.2L\text{”}]//Make = \text{“}Toyota\text{”} \tag{1.1}$$

It can be represented into a tree with root element 'Car' that has a child element named 'cc' having a content of "2.2L" and has a descendant element named 'Make' that has a content of "Toyota". This tree is called the 'Twig query' pattern for the XQuery expression of Equation 1.1. Figure 1.3 shows the twig query pattern.

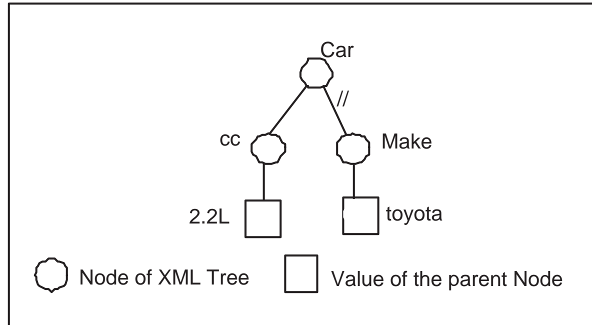

Figure 1.3: Example of a Twig query

Twig queries [9] (tree pattern queries) have been used to query the structural part [27] of XML documents. The structural join [7] and holistic twig join [11] algorithms that use twig queries have been developed to query native XML databases using the languages mentioned above. In our study we will use the twig query representation to specify a query pattern.

The fundamental problem of querying a database is to retrieve those elements that match the query. While searching the entire database for matching solutions is a trivial method, one can use several optimization aids such as structural summaries, for example, indexes and views [23, 15, 8]. We can also use cached pre-computations [13], semantic information in order to provide a quicker and much more efficient querying system.

Our Problem statement: Given a twig query pattern, we are required to determine if it exists in a given XML Document. Once the answer has been determined, upon the repeated execution of the same twig query, we should be able to answer the query with-out having to scan the complete document again. We are to answer such repeated queries using pre-computations. When the document is updated we must still be able to determine if a twig query pattern exists with out scanning the data again. This requires

incremental maintenance of the pre-computations stored. Additionally, with the usage of pre-computations we would like to obtain information such as the number of pattern matches that exist in the XML Document and some information regarding the extent of the query pattern that matches the document.

1.2 Thesis Contributions

Queries that determine if a pattern exists are known as '*boolean queries*'. The counts related to boolean queries can help in estimating statistics and characteristics of the document at hand. Boolean queries are useful in a publisher subscriber system [12], where a subscriber is sent only those publications that match certain conditions. Boolean queries can also be useful in secure dissemination of XML documents. The boolean queries can be used to check if the filtered secure XML document violates any security conditions. Generally, boolean queries are applicable for all situations that check for existence of a pattern.

Our first contribution is the development of an algorithm that pre-computes the result of the execution of a boolean query. A pre-computation can be defined as information that is collected and stored while searching for the solution the first time the query is executed. During the first search, some data is stored at various parts of the document. This ensures that a repeated query can be directly answered using the pre-computations. The idea of a pre-computation is effective as every-time a user queries for some data or to check if a pattern exists, the entire document does not have to be searched. The pre-computation is trivial as we only need to store a single entry specifying whether a query matches or not. The non-triviality arises from the fact that the document is subjected to updates. This leads to our second contribution. We provide the extensions to the pre-computation algorithm so that the pre-computed information can be maintained incrementally up-on the occurrence of updates without having to

re-compute the solution again. Our third contribution is an alternative algorithm that results in a larger number of pre-computations being stored. With this added information, one can also precisely determine the extent of partial query matches, further describing the nature of data. To see the importance, let us consider a simple illustration. Consider a query with two sub-trees to match. Suppose only one of two sub-trees of a query is matched in the document, then we retain that information. Now suppose the other sub-tree is added to the existing document, we are expected to immediately detect the presence of the solution without having to search for the sub-tree that has already been found. Using the second algorithm we can also obtain paths to all pattern matches. We give a theoretical complexity analysis of the algorithms followed by an experimental study of the performance of these algorithms on varied data-sets.

1.3 Thesis Organization

The rest of this thesis is organized as follows, in chapter 2, we present the related work, in chapter 3, we present some background information and describe the pre-computation model along with two pre-computation based query processing algorithms. It also includes a section on the complexity analysis for the various operations using these two algorithms. In chapter 4 we present the experimental setup and the experimental results obtained. Lastly,we provide our conclusion and directions for future research.

Chapter 2

Related Work

In this chapter we bring forth the various techniques that have been used for query processing and incremental maintenance. The problem of query execution over a XML database has been well studied, methodologies such as [7, 11, 19, 27, 20] have been implemented as solutions. The usage of structural summaries such as indexes have further optimized these solutions [23, 15]. In our study we are not trying to optimize these existing query execution methods, instead we are using a novel approach using pre-computations to answer queries.

This approach of using pre-computations appear similar to query result caching [13, 14, 29] and view materialization [8, 21]. The concept of the cache is that its contents are valid so long as the data is not modified. Upon updates it requires invalidations and re-fetching of results. In our scheme, we re-use the pre-computations on the occurrence of updates. The boolean queries used in this paper can be directly related to the domain of publisher subscriber system of XML documents [12]. A document is required to be published if it matches the pattern specified by the subscriber. Our scheme can be used in this model, even when the document is subjected to updates, we are able to determine if the document is required to be published without expensive re-computations.

Another core related work is in the area of schema validation of XML documents

[24, 10]. The problem in the case of schema validation is to determine whether the content of a given document matches a predefined DTD [2](schema). Here too, the complexity lies in determining, if a correct document still retains its correctness upon updates. The works of [24] and [10] can be referred to for solutions to this problem. While in our scheme we are trying to determine if a small tree pattern (twig) exists in the document, the schema matching problem can be thought of as validating the existence of many such twig patterns. [24, 10] too use pre-computed structures to enable incremental validation. In the remainder of this section we shall introduce some of the above mentioned methods and describe how our methodology resembles it or is inspired from it.

2.1 XML query processing using structural and holistic joins

Query processing using twig patterns on XML databases involves two essential steps, one, breaking down the twig query into a set of binary structural relationships and determine sets of data that match them and two, stitching together these basic matches to form the complete solution. For solving the first part of identifying the basic structural relationship matches, there have been several algorithms that have been proposed (refer to [11] for a complete list). Most of these algorithms rely on the labeling scheme used to identify the matching nodes. The positional representation labeling scheme [7, 11] can be used to identify parent-child and ancestor-descendant relationships present in an XML document in constant time. For the second part related to stitching together the matches, some efficient join ordering algorithms are required. In [11] the holistic twig join algorithm was proposed to reduce the impact of very large intermediate results produced in the first matching part, many of which are not part of the final solution.

In that paper, the authors proposed a method that would produce an intermediate match only if it was certain to be part of a solution. While the optimal execution of these algorithms can be aided with the use of indexes [17], it is still processed a query at a time and repeated joins need to be performed. The join ordering is a serious performance factor and detailed analysis and statistics of the nature of the database need to be gathered. Thus, while the simplicity of the algorithm appears to be in the determination of structural relationships, for it to be optimal, it requires several other performance aids.

Let us consider how these algorithms measure up to frequent updates. One critical issue is the support from the labeling scheme. As illustrated in the prime number labeling scheme [28], leaving gaps between labels is not a very feasible idea. Re-labeling is an expensive task. Also, as mentioned earlier the histograms and statistics about the data needs to be continuously updated and maintained upon updates. Lastly, frequent queries and similar queries are re-executed against the database unless this processing scheme is merged with some form of query caching.

In our algorithm, we de-couple the labeling scheme from the query processing. We also support optimal retrieval of solutions to frequent queries. In addition, our algorithm is designed to scale-up to dynamic XML documents. It must be mentioned that while our scheme targets boolean queries, the structural and holistic twig join algorithms are capable of retrieving the exact solutions. While in the experimental sections of [7] mention that tree-traversal algorithms have been considered inefficient, for boolean queries we show that the pre-computation based algorithms are indeed competitive and efficient.

2.2 Selectivity Estimation of Twig Queries

Given a XML document it is useful to understand the characteristics of the data. Information such as frequency of elements, patterns, join cost estimates etc. can optimize query processing. [16] uses a summary data structure to estimate the number of twiglets (small twigs) matches. It uses the individual estimate of twiglets to come up with an estimate for any twig query. This method uses a *correlated subpath tree* structure to represent the frequencies. This structure is maintained along frequently occurring sub-paths. While this estimation solution is part of the exciting set of approximation algorithms present in today's literature, it has not given any direction to how these structures are maintained upon frequent updates on an dynamic database.

In our algorithm, we provide the exact number of solution matches that are available at any subtree of the document. We also illustrate in the algorithm how these counts can be updated with a complexity of $O(\mathrm{d})$ where d is the depth of the tree. In addition, we can consider the counts of twig matches of a query providing an approximate result to another query similar to it. For example, if a new query Q_A is a sub-set of another query Q_B. By sub-set we mean that the twig query pattern Q_A that is to be matched is present as a sub-tree of another query Q_B. In this case, the lower bound of the query Q_A count is the count of the query Q_B.

2.3 Incremental validation of XML schema

Consider a XML database that conforms to a XML schema [2]. The XML Schema impose structural constraints on the structure of the database. When updates on the database occurs, one needs to check if any of the constraints are violated. Re-validation of the entire database for each update would be a very costly operation. Using pre-computations, this cost can be drastically reduced. The algorithms presented in [24, 10]

are examples of this method.

The problem we are trying to solve is a much simpler problem. While the entire schema could be thought of as a large set of twigs that must exist in the database. We are trying to determine if a solution to such a query exists. The former problem is compounded by the fact that there could exist some nodes that match a query and but is a partial match of the query. This may imply a violation of the schema. In a boolean query one occurrence of a solution is enough for satisfy the query, where as in the schema validation scheme, every occurrence of a node that belongs to a query implies that a complete solution using that node is to be found.

2.4 Discussion

The XML query processing methods such as Structural joins [7] and Holistic twig joins [11] are essentially used to retrieve the set of matching solutions. These require indexes on all the elements to be matched and perform a set of join operations to return the results. In contrast we are only trying to determine solutions to boolean queries. Thus, ours is a simpler problem. Determining the exact number of counts of a pattern in a given document is part of actually searching the query. [16] has given us an idea on how pre-computed values on twiglets(small queries) are used in conjunction to determine an estimated result set size. In our case, we are actually trying to determine exact counts of the number of solutions. This directly implies that, if the count is greater than zero, the boolean query returns true indicating a pattern match otherwise returns false indicating that there is no solution present in the document. The incremental validation of XML schema [24] and[10] has to ensure that every schema rule is completely matched in the document. It also has to ensure that if some nodes that match a query exists, then it must be part of a complete solution. This problem we are trying to solve only checks to see if one such solution exists. In the following chapter we describe how our approach

finds solutions to the boolean query.

Chapter 3

Querying using pre-computations

3.1 Preliminaries

Finding all matches of a query twig pattern in an XML database is a core operation in XML query processing, both in relational implementations of XML databases and in native XML databases. Given a query twig pattern Q and an XML document D, a match of Q in D is identified by a mapping from nodes in Q to nodes in D, such that: (i) query node matches the corresponding database nodes, and (ii) the structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes.

A boolean query is a query that determines if the query pattern matches the document. The answer to the boolean query Q with n nodes to match is stored at the root of the document D. The root of document D also contains the count of matching solutions to the query Q. In this thesis, we consider the boolean twig pattern matching problem: Given a query twig pattern Q, and an XML database D , compute the answer to Q on D that represents the solution indicating whether the pattern exists and if it exists the total number of solutions available in D, but not the actual data nodes. While the boolean query can express any type of query, we will omit those queries that require

ordering and contains repetitions of element nodes. We however give some direction how these types of queries can be handled in the conclusion of this thesis. As an extension we also determine the maximal extent to which solutions are present in the database. Intuitively, partial matches of queries can contribute to statistics too. Also, we could devise a method to use these partial matches by checking if the solution of a new query is present as a subset of the result of a previously executed query. Figure 3.1 is an example of a twig query that is used to match all Red Honda SUVs of the XML document shown in Figure 1.1.

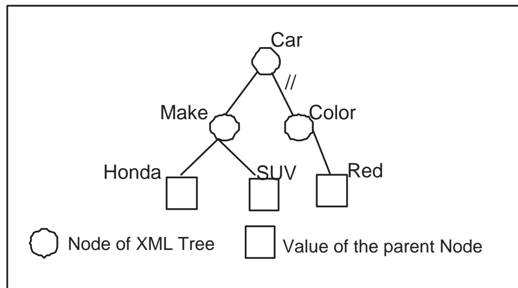

Figure 3.1: Another example of a Twig query

Consider, for e.g., the query twig pattern in Figure 3.1. The nodes in D that match the root of Q('Car') stores the number of pattern matches that exist using its sub-tree. This information is also sent to the root of the document D. After the pre-computation phase, if query Q is re-executed, the root of the document D contains the answer to Q.

3.2 The pre-computation model

The objective of the pre-computation is to determine if the query match can be found in the document and to store that information. Thus we need to define how this search is to be performed and what information needs to be pre-computed and stored.

The pre-computation is carried out by executing a recursive procedure in a depth first manner over the XML tree. After a complete recursive traversal of the document, all nodes that participate in any solution of the query will store information about

that query. Figure 3.2 illustrates how a recursive process can be used to determine the existence of the twig query match.

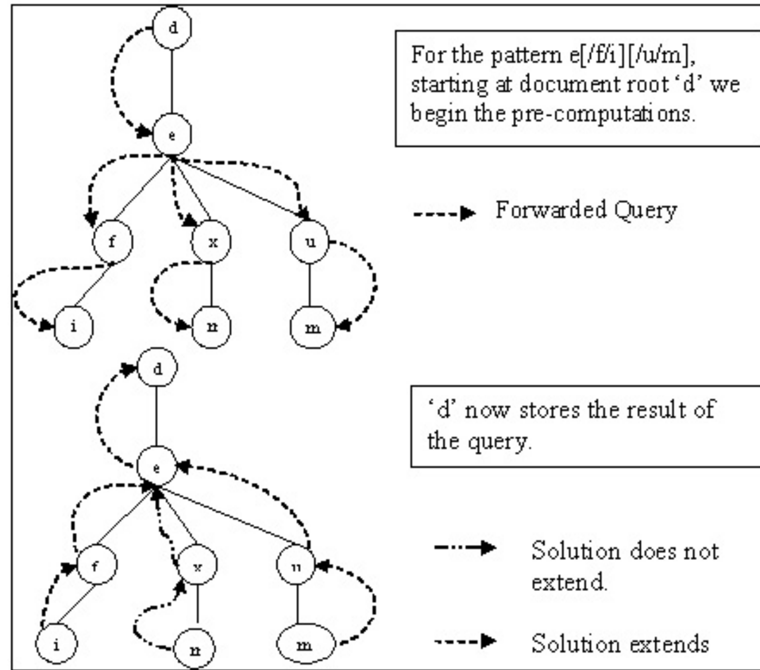

Figure 3.2: Recursive procedure to check if a solution exists.

Given a boolean query(Q), we are trying to determine at the each node(say N), the maximal solution of the query that is matched by node N's sub-tree(*Sub-tree(N)*). By *Sub-tree(N)*, we mean node N, its children nodes and all its descendants. *Figure 3.3* shows an example of a node storing the maximal subtree.

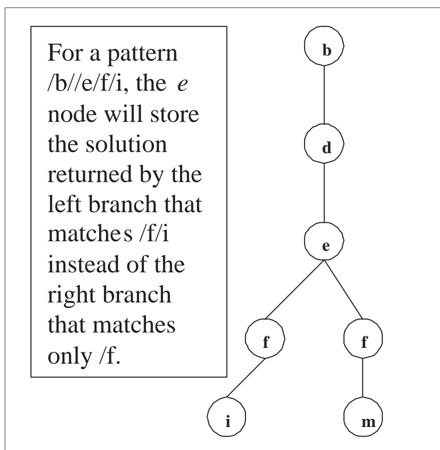

Figure 3.3: Example of node storing the maximal subtree match

At the root node of the document, we store the result of the query, that is whether

the complete twig query pattern has been matched by this document. At each of the nodes of the document that matches the query, the count of the total number of complete sub-tree matches for each descendant position of the query is also stored. This count helps us determine the total number of solutions that match the query. We illustrate this idea using the following example. Consider the XML tree and the query shown in Figure 3.4. We have shown the state of the document tree before and after the pre-computations. We notice that the sub-tree of the nodes that are marked with a 'C' contain complete sub-tree matches of the query from the position it matches, where as nodes that are marked with a 'P' only match the query Q. For example, The sub-tree of the node of the document with the tag 'cc' that has been marked with a 'C' contains the complete subtree of 'cc' of query Q. We also notice that the root of the document contains a pre-computed value indicating if a solution to the boolean query Q exists.

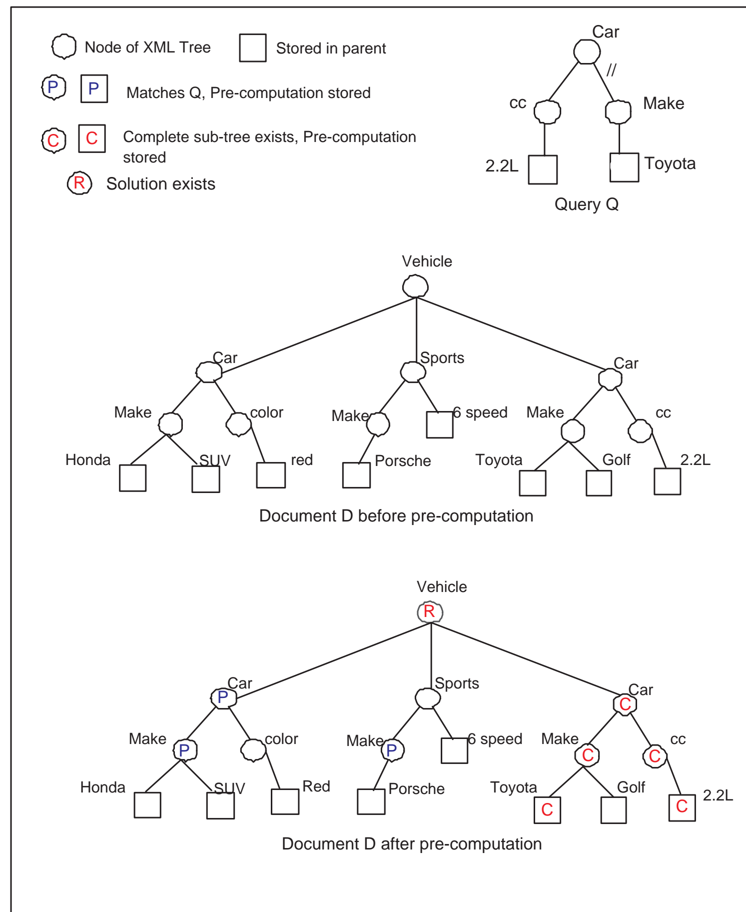

Figure 3.4: Pre-computation of an XML document for query Q

The use of pre-computed information not only lies in answering repeated queries. It can be effectively used to re-compute the pre-computations upon updates without having to scan through the entire document again. This is determined by the kind of information that is pre-computed and stored at the nodes summarizing the structure of the entire document. Thus the amount of pre-computed information stored greatly influences the effort required to re-compute information upon updates. In our model we are presented with two choices.

- Only nodes that participate in a solution store any pre-computations, we develop this into the **NodeMatch** model.
- Apart from matching nodes, all the nodes that lie in a path of a solution (intermediate nodes) store information, this is modelled into the **PathMatch** technique.

The difference in these two methods appear when updates operations are performed. If the intermediate nodes store information then re-computing the new state is easy as all information required to re-compute will be present in the level at which the updates occur. In the case of only participating nodes storing pre-computations, certain searches of pre-computed information in the sub-tree affected are required. However, both these methods are better than having to search the entire document again, This advantage is gained by paying the cost of extra storage for the pre-computations.

In summary, given a node that matches the query(Q[1..n]) at Q_i we need store some pre-computed information that captures this information. we store a data-structure representing the maximum matching sub-tree of Q_N. It is maximal in that all the possible children and descendant matches to the query is stored in the pre-computation. Additionally, if the query contains descendant positions to be matched, we store the count of the number of matches for each such complete descendant sub-tree [1]. In section

[1] Explained in the algorithm

3.3.1, we describe the structure of the pre-computed data.

3.3 Definitions and data structures

The *Pre-computation phase* is a recursive procedure that is executed in order identify the nodes at which pre-computations are to be stored. This is done by calling the method *find_pattern* (Figure 3.8). This phase is common to both NodeMatch and PathMatch. After the complete recursive cycle, all nodes that completely or partially participate in any solution of the query will store information about that query.

3.3.1 The probe:A pre-computation data structure

The *probe* is a data structure that is used to collect the information regarding the participation of nodes in the solutions. The probe contains in it two arrays that are used to represent the query tree. They are used to mark the nodes of the query that are matched by the sub-tree of node N at which the probe is stored and the number of such matches. These two arrays are used as follows,

- The first array is a bit array that is used to indicate the positions at which the sub-tree of N match the query.
- The second array is an integer array that is used for storing the counts for each matching query position.

The probe also contains a count of the number of complete sub-tree matches that exist from N. For example, suppose N matches the query(Q[1..n]) at Q_i, then the counter stores the number of complete subtree matches of Q_i that can be found in the subtree of N. The probe also contains two lists, a next_position_child list and a next_position_desc list. The next_position_child list contains the next children that the probe needs to find to extend its solution. Similarly, the next_position_desc list has the list of descendants

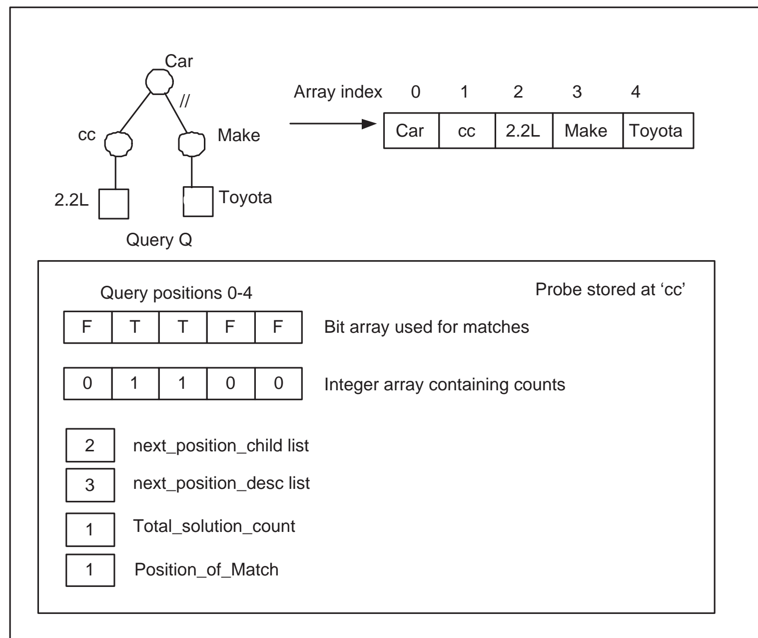

Figure 3.5: The structure of a stored probe

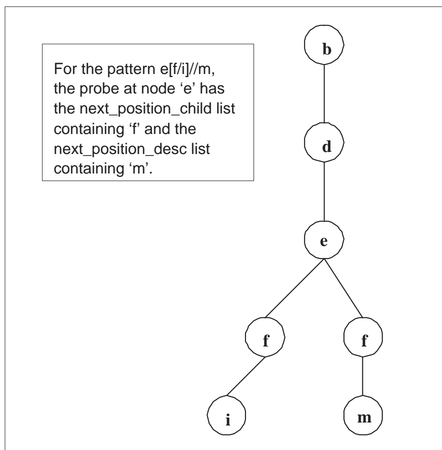

Figure 3.6: Use of the two lists in the probe structure

that are to be matched for the solution to be complete. For example, during the pre-computation phase, the state of these lists is shown in Figure 3.6. For the document and query shown in Figure 3.4 consider the node with element tag 'cc' that has a content of '2.2L'. The probe that is stored at 'cc' is shown in Figure 3.5.

The number of matches of each of these nodes in the list is stored in the integer array mentioned above. This count is used to determine the total number of solutions that exist at a subtree. For example, consider the query //Car[//Red]/Honda. For a node N that matches 'Car', its next_position_child will have 'Honda' and the count

will be the number of children of N that match 'Honda' and its next_position_desc will have 'Red' and the count for the 'Red' list will have the number of 'Red' descendants N has in its sub-tree. The probe contains a value called the position_of_match. This value represents the position at which the node matches the query. For example, the probe shown in Figure 3.5 contains this value as '1' because 'cc' is in position 1 of the array used to represent the query. Lastly, the total_solution_count is an integer that stored the total number of pattern matches of the sub-tree of the query starting at the position_of_match. For a node that matches the root of the query, this value contains the total number of complete solutions to the query that is present in the entire sub-tree of the node.

When we are required to extend this model to support queries that contain repetitive elements, we could use an array of probes, each indicating the corresponding position of match.

3.3.2 Representation of XML query

The given XML query Q is modelled into an XML tree named Q_t, Thus, the solution to the query Q, lies in determining if Q_t is present as a pattern of nodes of D. We also label the nodes of Q_t using the range numbering scheme as described in [11]. If Q_t is traversed using a pre-order traversal and written into an array named $A_q[0..n]$ where n is the size of the number of nodes in Q_t, Given a node of this query tree labeled Q_i, we can determine its entire subtree using the indices obtained using the start and end labels of Q_i. This property can be used to check if a complete sub-tree exists. The order of elements as provided by the pre-order traversal is used to store the query Q in the probes mentioned earlier.

3.4 Overview of NodeMatch and PathMatch algorithms

NodeMatch and PathMatch can be used to process a given boolean twig query against an XML document. As described in section 3.2, NodeMatch stores pre-computations only at the nodes that match the query where as PathMatch stores probes at nodes that match the query and along the path from the root of the document to each of these nodes that match the query. Thus, an important difference that exists between the two models is in the number of probes stored. These additional probes stored will help in faster incremental maintenance of updates. These also allow us to trace the path from the root of the document to every solution that exists in the document. Figure 3.7 shows an example query and the probes stored in a part of an XML document. The key intuition behind NodeMatch can be explained as follows. The entire document is scanned once, resulting in pre-computations being stored at all the required nodes. Additionally, the root of the document stores the result of the query. When the document is subjected to updates, the pre-computations at the nodes that lie along the path from the node at which the update is done to the root of the document are updated to reflect the new state. Updates at a node that does not store a probe can require searching its sub-tree this is a potential performance bottleneck of NodeMatch. In contrast, with PathMatch if there is a solution in the sub-tree of a node then it must store a probe. This avoids searching the sub-tree which could be computationally expensive. The idea with PathMatch is to avoid searches down the XML document tree, and restricting all operations to work up the XML document tree along the path to the root. The complete comparison of NodeMatch and PathMatch is provided in section 4.3.

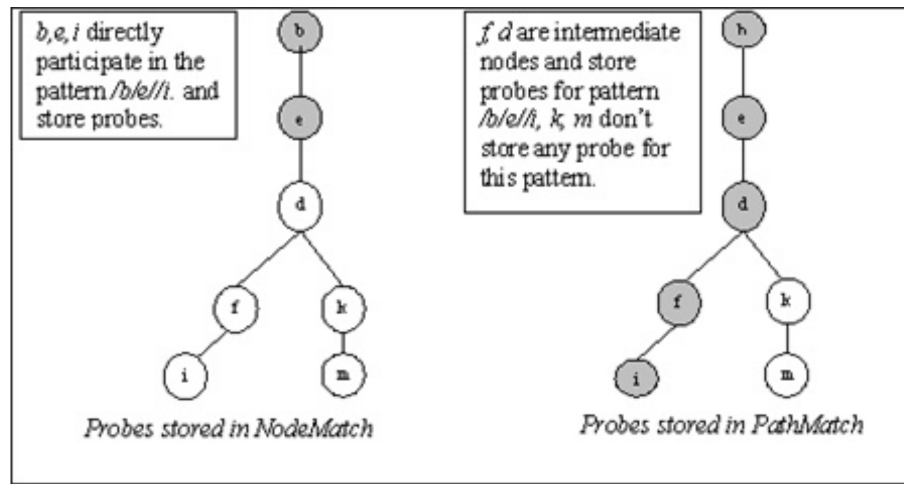

Figure 3.7: NodeMatch and PathMatch storing probes

3.5 The NodeMatch Algorithm

The NodeMatch algorithm, stores minimal pre-computations that help in quick response to queries in addition to supporting incremental maintenance of the pre-computations when subjected to updates. If updates are not required to be supported, the solution is trivial and just needs a one time traversal. However, the objective here is to be able to support updates and incrementally maintain the pre-computed information. NodeMatch stores pre-computations only at nodes that directly match the query and at the root of the document. An example of a document that has pre-computed the solutions to a query Q using NodeMatch is shown in Figure 3.4. The first phase is to perform the traversal of the document and determine all these nodes that need to store the pre-computations. In addition, we also determine all the existing solutions, its count and the different partial matches. We describe the details in the following sub-section.

The Initial pre-computation phase of NodeMatch:

We introduce below the procedure to find all results of a Query Q and pre-compute the information that is going to be used in later queries and during updates. Given a query Q, a document D that has been parsed into a tree representation with root R_t,

the procedure *find_pattern* (Figure 3.8) is executed. This results in all the nodes that participate in the solutions of this query Q storing the pre-computations. If the root of the query Q_r is matched at N_r, then N_r will contain the number of solutions to Q that exist in sub-tree(N_r). For a single instance of the execution of *find_pattern* the following steps are carried out.

If the node N matches a node Q_n of the query, the different possibilities that can occur are discussed below as cases and follow the if-else sequence of the algorithm.

1. Case 1: Node matches the query and is the first node to match. Create a new probe and initialize it using the function *create_probe* (Figure 3.9). The *create_probe* function marks a nodes presence and populates the next_position_child/desc lists using the query.

2. Case 2: From the received probe, the node could extend a solution.

    - Case 2a: Node is one of the next children to be matched for the probe. Mark its position in the probe, set it to be stored , also set the next positions to be matched using the *set_next_position* method (Figure 3.11) .

    - Case 2b: Node is one of the next descendants to be matched for the probe. Mark its position in the probe, set it to be stored , also set the next positions to be matched using the *set_next_position* method (Figure 3.11) . Cases 2a and 2b can be merged into one condition as: if node matches either the next_position_child/desc. But for now has been retained separately for extension purposes.

    - Case 2c: The current probe is not extended by this node match, but as its descendants (if any) can match, it may have to be retained. If it has any descendants to match in its next_position_desc list It is marked to be forwarded. The next_position_child list is cleared. Else It is marked as not_forwarded.

3. Case 3: The node is a match, but does not extend the previous probe. (i.e. the *new_probe* flag is still true) Create a new probe and initialize it using *create_probe* (Figure 3.9).

4. Case 4: If the node does not match any node in query pattern, then probes that do not have any descendants to be matched can be stopped from propagating any further. For this purpose the *prune_probe* method (Figure 3.10) is used. The *prune_probe* method checks whether the probe's next_position_desc list is empty, if so, it is marked as *not_forward* otherwise set it is set to be forwarded and next_position_child list is cleared.

As per the current logic only one new probe can be created, this is because of the assumption that a node can match only at one position in the query pattern. It must also be noted that only one position can be extended too, thus at any point only one probe is stored per query. As an extension, if the query pattern is permitted to have multiple occurrences, then, new probes could be created for each new position for which no solution currently extends.

Now that we have determined whether the probe is be forwarded and if a new probe is to be created, we can create the final set of probes to be forwarded to its children using the current probe that has been marked to be forwarded. This functionality is provided by the *forward_to_next_level* method (Figure 3.12). It is further explained below.

For each child of N, execute *find_pattern* using the probe thats marked to be forwarded. From the returned set of probes, we compute the probe that need to be stored at this node by merging the multiple subtree matches from different children into information in a single probe. This is done using the *find_best_match_and_store* method (Figure 3.13). This also involves maintaining the counts. It returns the probe that need to be returned to the parent node of N.

```
1: Function find_pattern(Node N, Query Q, Probe P)
2: initialize flag new_probe to true
3: if N matches any node Q_n of Q then
4:    {Case 1:}
5:    if probe is empty then
6:       Call create_probe(N, Q, Q_n)
7:    else
8:       Initialize flag new_probe to true {Case 2:}
9:       if ( N = ANY Pr→next_position_child ) then
10:         Update Pr to include N {Case 2a:}
11:         Mark Probe to be stored
12:         Call set_next_positions(Pr, N, Q, Q_n)
13:         Set new_probe to false
14:      else if (N = '//' Match in Pr and N present in Pr→next_position_desc) then
15:         {Case 2b:}
16:         Update Pr to include N
17:         Mark Probe to be stored
18:         Call set_next_positions(Pr, N, Q, Q_n)
19:         Set new_probe to false
20:      else if (new position Match) then
21:         {Case 2c:}
22:         {If the current probe has any descendants to match, the current probe can
            continue to find descendants}
23:         if Pr→next_position_desc is empty then
24:            Mark probe Pr as not_forward
25:            Call set_next_positions(Pr, N, Q, Q_n)
26:         else
27:            Mark Pr as forward
28:            Set Pr→next_position_child to empty
29:         end if
30:      end if
31:      if (new_probe is true) then
32:         Call create_probe(N, Q, Q_n) {Case 3:}
33:      end if
34:   end if
35:   { end of check if empty probe}
36: else
37:   Call prune_probes() {N does not match any Node}
38: end if
39: Probe finalProbe = forward_to_next_level(N, Q, P)
40: RETURN finalProbe
41: End of function find_pattern
```

Figure 3.8: Function find_pattern()

```
1: Function create_probe(Node N, Query Q, QueryPosition Q_n)
2: Create Probe Pr
3: Set position of match in Pr to be Q_n
4: Mark Pr to be stored
5: Call set_next_positions(Pr, N, Q, Q_n)
6: End of function create_probe()
```

Figure 3.9: Function create_probe()

```
1: Function prune_probe()
2: for each probe Pr in P[ ] do
3:    if Pr→next_position_desc is empty then
4:       Mark probe Pr as not_forward
5:       Call set_next_positions(Pr, N, Q, Q_n)
6:    else
7:       Set Pr→next_position_child to empty
8:       Mark Pr as forward
9:    end if
10: end for
11: End of function prune_probe()
```

Figure 3.10: Function prune_probe

```
1: Function set_next_positions (Probe Pr, Node N, Query Q, QueryNode
   Q_n)
2: {check if N is a descendent waiting to be found}
3: if N IN Pr→next_position_desc  then
4:    remove N from Pr→next_position_desc
5: end if
6: if Q_n is leaf of Q and Pr→next_position_desc is empty then
7:    Set Pr to not_forward
8: end if
9: if Pr set to not_forward then
10:    set Pr→next_postions_child to empty
11:    Return
12: end if
13: initialize Pr→next_position_child to empty
14: {From Q, set the next children positions to be found}
15: for all children Q_nc '/' of Q_n do
16:    Add to Pr→next_position_child Q_nc
17: end for
18: {From Q add the next descendants of Q_n to be found}
19: for all descendants Q_nd '//' of Q_n do
20:    Add to Pr→next_position_desc Q_n
21: end for
22: End function set_next_position
```

Figure 3.11: Function set_next_position()

The function *set_next_positions* (Figure 3.11) populates the next_position_child/desc lists in addition to providing some minor processing logic. Its inputs are the current node N, the probe Pr, the query Q and the position of the current match Q_n

If Q_n is a descendant node match and is currently in the next_position_desc list it is removed as it need not be matched now for extending the current probe. However, it is important to realize that, if the current node N was not part of the solution, another node N_d in the subtree(N) could match Q_n. Thus, to arrive at the correct number of solutions available at a node, the method *find_best_match_and_store* (Figure 3.13) contains logic that maintains counts for number of matches for each of these complete subtree matches for descendant positions in Q.

Suppose the node matched a leaf node of the pattern, then no further propagation is required if its next_position_desc list is empty.

If the probe is not to be forwarded then clear its lists, other wise this method is used to fill the next positions that need to be matched. Firstly clear the next_postion_child list. Each child node of node Q_n of pattern Q is added to the next_position_child list of Pr. Each descendant of Q_n that is to be matched is added to the next_position_desc list of Pr.

The function *find_best_match_and_store* (Figure 3.13) collects the probes that N sent to its children, and tries to find out the best possible extension to the solution. The intuition here is that, if a child node extends a larger subtree of the solution, retain that as the best possible match, which could later result in a complete solution. The probe 'finalProbe' will be returned to the parent of this node. This probe contains the count (*desc_position_count*) of complete subtree matches for each descendant position in query Q. The desc_position_count counters are stored in the array representation of the query tree in the probe.

From the list of forwarded probes, we need to check if solution extensions ex-

```
1: Function forward_to_next_level(Node N, Query Q, Probe P)
2: Create a probe MPr
3: Copy probe P marked as forward to MPr
4: for each child N_c of N do
5:    retProbes[c] = find_pattern(N_c, Q, MPr)
6: end for
7: Probe finalProbe = find_best_match_and_store(N, MPr, retProbes[ ])
8: RETURN finalProbe
9: End of function forward_to_next_level
```

Figure 3.12: Function forward_to_next_level()

```
1: Function find_best_match_and_store (Node N, Probe Pi, RetProbes[ ])
2: Create a probe called finalProbe
3: initialize next_child_counts[ ], next_desc_counts[ ], total_solution_count of finalProbe
   to zero.
4: for Each Q_c IN Pi→next_position_child do
5:    Set childFlag to true {childFlag to indicate extension of child solution}
6:    Call check_for_extension(Pi, Q_c, childFlag)
7: end for
8: for Each Q_d IN Pi→next_position_desc do
9:    Set childFlag to false
10:   Call check_for_extension(Pi, Q_d, childFlag)
11: end for
12: Call compute_counts_and_merge(Pi, Q_n)
13: if Pi marked to be stored then
14:   Store the Pi→tree, Pi→desc_positions_count[], next_child_counts[] and
      Pi→total_solution_count into finalProbe
15:   Store finalProbe at N.
16: end if
17: RETURN finalPRobe
18: End function find_best_match_and_store
```

Figure 3.13: Function find_best_match_and_store()

ist. The *check_for_extension* function (Figure 3.14) performs this task. The childFlag parameter determines if we are checking for an extension of a next_position_child or next_position_desc.

In *check_for_extension* function (Figure 3.14), given a Probe P_i , check all the returned probes from each child. Suppose, the return probe of child 'a' extended the solution using next_position_child '1', and so did child 'b', then depending on whether 'a' or 'b' has a more complete solution, the matching information from it is copied.

Also suppose the return probe of child 'a' extended the solution using next_position_child

```
1:  Function check_for_extension(Probe P_n, Query Node Q_x, Flag childFlag)
2:  for Each RP_i IN RetProbe[1..n]→probe do
3:     {RetProbes[i] or RP_i refers to the probe of the i^th child of N}
4:     if Q_x matched in RP_i then
5:        {This probe has been extended by RP_i}
6:        if number of matched nodes at Subtree(Q_x) in RP_i > Matched nodes in Sub-
          tree at Q_x of P_n then
7:           Copy all matched nodes of the sub-tree(Q_x) of RP_i into P_n
8:        end if
9:        if Subtree(Q_x) in RP_i is complete then
10:          if childFlag == true then
11:             Increment next_child_counts[Q_x] by 1
12:          else
13:             Increment next_desc_counts[Q_x] by RP_i → total_solution_count
14:          end if
15:       end if
16:    end if
17: end for
18: End of function check_for_extension
```

Figure 3.14: Function check_for_extension()

```
1:  Function compute_counts_and_merge(Probe Pi, Query Node Q_n)
2:  if Pi is a complete match at Q_n then
3:     {Compute the total number of solutions}
4:     if Q_n is a leaf then
5:        Set total_number_solution_count to 1
6:     else
7:        Set Pi→total_solution_count = product of all next_child_counts[],
          next_desc_counts[]
8:     end if
9:  end if
10: End of compute_counts_and_merge
```

Figure 3.15: Function compute_counts_and_merge()

'1', and another child extended the same probe using next_position_child '2' or next_position_desc 'x' , then this represents a twig in the query, hence is merged.

Regarding the counts, we maintain a few counters, one of them is the total_solution_count. This counter stores the total number of complete subtree(Q_n) matches that can be found in the subtree(N). Another set of counters *desc_position_count* are used to store the total number of complete descendant subtree (*i.e. if query Q has a descendant 'x' which has its own subtree, then this counter stores the total number of matches for subtree(x)*) matches of the query Q that have been found at N. The desc_position_count values are propagated until the root of the document.

If the probes obtained from its children contains the entire subtree from its position, the total number of solutions is equal to the product of the non-zero counts available at each of the next_position_child and the desc_position_count for each next_position_desc. These calculations are performed by the *compute_counts_and_merge* method (Figure 3.15).

The nodes that match the root of the query will now store the information indicating if a complete pattern can be found in its subtree and the corresponding counts. All complete solutions are propagated towards the root of the document and from which the existence of a solution and the complete count can be obtained.

3.6 Incremental maintenance of NodeMatch

In this section we discuss the procedures to incrementally maintain the pre-computations that have been stored using the NodeMatch technique. The following types of updates can occur in any XML database.

1. Deletion of an entire subtree of N

2. Deletion of a partial(intermediate) subtree of N

3. Insert of subtree at a leaf node or as new child

4. Insert of subtree at intermediate nodes.

5. Updates on existing nodes.

. For the current discussion we will omit cases 2,4 and 5.

The essential operations involved in incrementally maintaining the pre-computed values are identifying the nodes affected, the recalculations to be performed and the propagation of the recalculations. The efficiency of an incremental maintenance scheme lies in re-using presently stable information and re-computing the new state with minimal re-calculation. We shall now present the algorithms to support the updates operations.

3.6.1 Insertion of a complete sub-tree using NodeMatch

The insert operations have implication that new solutions to the queries exist in the new subtree and also include the possibility of extending other partial solutions present in the existing document. While adding a single node as a leaf node is as simple as checking if it is present in its parent's next_position_child/desc list or any of its ancestor's next_postition_desc list, the insert of entire subtree can be handled a little differently than adding one node at a time. We can execute the *find_pattern* method (Figure 3.8) against the new sub-tree and collect the entire pre-computation information at the root of the new sub-tree. This is stored using a probe called the *insert_probe*. Now the addition of this sub-tree with pre-computations is similar to adding a single leaf node. The details are given the algorithm describing the *insert_subtree* function (Figure 3.16). Once the parent re-computes its status, it must correct the information stored in its path to the root by passing the *insert_probe* to all those nodes. This is done using the *correct_parent_increment* function (Figure 3.17).

The *correct_parent_increment* function (Figure 3.17) essentially recomputes the to-

1: **Function insert_subtree(Tree T)**
2: Call find_pattern(T, Q)
3: Set N_r to root of T
4: Initialize desc_position_cnt[] to zero
5: **if** N_r matches a next_position_child Q_c of Parent(N_r) **then**
6: **if** Subtree(Q_c) in N_r is complete **then**
7: SET up_forward to true
8: **end if**
9: **end if**
10: Copy to insert_probe.desc_position_count the values in desc_position_count[] of N_r
11: **if** N_r matches a next_position_desc Q_d of Parent(N) **then**
12: Copy to insert_probe.desc_position_count[Q_d] the value in total_solution_count of N_r
13: **end if**
14: Call Parent(N).correct_parent_increment(insert_probe, up_forward, N)
15: **End of function *insert_subtree***

Figure 3.16: Function insert_subtree() for NodeMatch

1: **Function correct_parent_increment(insert_probe, flag up_forward, N)**
2: **if** up_forward is true **then**
3: increment next_child_count(N) by 1
4: **if** *total_solution_count* == 0 **then**
5: {No complete solution found yet}
6: Copy sub-tree Q_n into the stored probe at N
7: **if** new child does not complete parents subtree **then**
8: set up_forward as false
9: **end if**
10: **end if**
11: **end if**
12: **for** each descendant position Q_d of Q **do**
13: **if** $desc_pos_match[Q_d] == 0$ **then**
14: Copy probe subtree of Q_d stored at N
15: **end if**
16: increment desc_position_count[Q_d] by insert_probe.desc_position_count[Q_d]
17: **end for**
18: calculate new total_solution_cnt
19: update insert_probe if new solutions are found.
20: **if** N not root **then**
21: Call Parent(N).correct_parent_increment (insert_probe, up_forward, Current Node)
22: **end if**
23: **End of function *correct_parent_increment***

Figure 3.17: Function correct_parent_increment() for NodeMatch

tal_solution_count at each node on the path of the new sub-tree till the root of the document. if the up_forward flag is true, it corrects the counts corresponding to the next_position_child list. It also determines if this has lead to the probe at N_p becoming a complete sub-tree. Otherwise, at N_p, the counts corresponding to the complete descendant matches in the new sub-tree are added to the current counts. Finally the new total_solution_count is computed and stored. If the new data has lead to new solutions at N_p, these new solutions are added to the insert_probe. The recursive call continues till the root of the document is reached.

3.6.2 Deletion of a complete sub-tree using NodeMatch

The delete_subtree receives a Node N as input, It needs to delete all the nodes in its sub-tree and also update all the probes in its parent and ancestors that have solutions extended by the sub-tree of N. The key intuition is that, only descendent matches need to be propagated upwards in the subtree being deleted, because the parent-child solution can be extended only by the root of the sub-tree being deleted. The idea is as follows, we will only propagate complete descendent subtree matches. The count of matches for the descendent position in the query are sent to N's Parent/Anscestors. This number will then be deducted from the counts stored at those nodes. If the new count at any of N's Parent/Ancestor is zero, then the descendent match and its subtree in the stored probe is deleted. Also if the possible parent/child solution is deleted then the flag up_forward is used to inform the parent of N.

The function *delete_subtree* (Figure 3.18) creates a probe called *delete_probe*. This probe contains the counts that represents the number of complete subtree matches of the descendants that are present in the deleted position of the subtree. For each child we execute *find_desc_matches* (Figure 3.19).

The *find_desc_matches* function (Figure 3.19) recursively gets the counts for the

```
1: Function delete_subtree(Node N)
2: create an empty delete_probe
3: Set up_forward to false
4: {Try to determine the number of complete descendant matches for each possible
   descendant position of Q}
5: for each child N_c of N do
6:    find_desc_matches(N_c, delete_probe)
7: end for
8: if N matches a next_position_desc Q_d of Parent(N) then
9:    Copy to delete_probe.desc_position_count[Q_d] the value in total_solution_count of
      N
10: end if
11: if N matches a next_position_child Q_c of Parent(N) then
12:    if Probe at N matches complete subtree(Q_c) then
13:       Copy to delete_probe.child_position_count[Q_c] the value in total_solution_count
          of N
14:       Set up_forward to true
15:    end if
16: end if
17: Call Parent(N).correct_parent_decrement(delete_probe, up_forward, N)
18: End of function delete_subtree
```

Figure 3.18: Function delete_subtree() for NodeMatch

number of complete subtree matches of any descendent position Q_d of Q. This is obtained by finding the first probe on each path from the root of the subtree being deleted to its leaf. The counts are copied into the *delete_probe*. These counts need to be deducted in from the total solution counts of N's parent/ancestors.

The *correct_parent_decrement* function (Figure 3.20) updates the counts stored in the parent nodes and the ancestors. If the upward_flag is true it implies that its child node (The root of the sub-tree being deleted) no longer extends its solution. The next_child_count for N is decremented by one. If the count reaches zero that subtree is deleted from the probe stored. The parent also decrements each descendant count present in the delete_probe and similar to the case of the child, if any descendant count reaches zero, its information is deleted from the probe. Lastly the total_solution count is recalculated. Now forward the new information and status of the solution counts to its parent. This propagates till the root.

1: **Function find_desc_matches (Node N, Probe delete_probe)**
2: **if** N is leaf and has no probe **then**
3: RETURN
4: **end if**
5: **if** N has a stored probe **then**
6: Copy to desc_position_cnt[Q_P] the value in total_solution_count of the stored probe
7: RETURN
8: **else**
9: **for** each child N_c of N **do**
10: set delete_probe.desc_position_count[] to find_desc_matches(N_c, delete_probe)
11: **end for**
12: **end if**
13: **End of function *find_desc_matches***

Figure 3.19: Function find_desc_matches()

1: **Function correct_parent_decrement(Probe delete_probe, flag up_forward, Node N)**
2: **if** up_forward is true **then**
3: **if** $(child_pos_cnt(N) - 1) == 0$ **then**
4: make total_solution_count as zero
5: delete from probe subtree of N
6: decrement next_child_count(N) by 1
7: set up_forward as false
8: **end if**
9: **end if**
10: **for** each descendant position Q_d in Q **do**
11: decrement desc_position_count[Q_d] by delete_probe.desc_position_count[Q_d]
12: **if** $desc_position_count[Q_d] == 0$ **then**
13: delete from probe subtree of Q_d
14: **end if**
15: **end for**
16: calculate new total_solution_cnt
17: if the deletion has resulted in other solutions becoming invalid, add these counts to delete_probe.
18: **if** N not root **then**
19: Call Parent(N).correct_parent_decrement(delete_probe, up_forward, Current Node)
20: **end if**
21: **End of function *correct_parent_decrement***

Figure 3.20: Function correct_parent_decrement()

3.6.3 Complexity analysis of NodeMatch algorithm

The pre-computation phase:

The pre-computation process involves traversing the entire document and hence its complexity would be determined by the depth 'd' of the document and the fan out 'f' of each node. Its also determined by the number of times the iterations in find_pattern are executed. If a single descendant node is to be matched, the number of solutions that need to be extended at the worst case is still equal to 1.

Let's now consider a query Q with q_n nodes and x descendant nodes and average of q_c child nodes at each level. The overall computation complexity of the find_pattern method can be expressed as $d*f*(f*q_n)$. The '$d*f$' component appears from the depth first search, the '$f*q_n$' for the analysis of the probes collected from a nodes children. The space complexity of the find_pattern method can be analyzed from two aspects, run time and actual storage. The run time memory in the worst case will be the size required to store N instances of probes. This occurs if all nodes of the document match the query. In general for a set of n nodes of a document D, a query with n_q nodes can at-most store n probes. Optimizations can be made, so as to not store zero counts, and store only the subtree of Q that matches N)

The final storage at each node that matches a node in the query will be the size of the probe S_p. Given the document D, if there are m nodes that match the query plus the one probe at the root, then we can expect the size required to be at-least $(m+1)*S_p$.

Complexity of the incremental maintenance

Insertion of nodes/entire subtree:

Case (i) only one node S_r is being added or a subtree rooted at S_r being added that does not extend any descendant solutions. If node S_r matches Q as a child position match, then a maximum complexity of depth 'd' node operations is involved. This includes,

the calculation of the new total solution count in case it's a complete extension to the solution and the updates of all nodes on its path till the root.

Case(ii) Adding a complete subtree that includes descendent matches. The complexity for update involves one complete search in the subtree S_t, and upward propagation cost of d, where d is the level of Parent(N) in the tree. At each node, the time spent is equal to the number of its next_position_child/desc stored and includes the time spent to recalculate the total_solution_count and the update of probes.

Deletion of a node/entire subtree:

Let us consider two cases, one where the sub-tree being deleted does not have any descendant matches and the other with descendant matches.

Case(i) : Sub-tree has only parent-child extensions to solution. In this case the complexity is restricted to the node N at which the deletion takes place. The parent(N) will simply have to check if N extends any of its complete solutions, if so then it must reduce its count in next_position_child by 1, leading to a complexity O(1) operation and re-computes the total_solution_count which has a complexity of $(x+q_c)$, where 'x' is the number of descendants and q_c, the number of child nodes in query Q. The parent needs to inform its ancestors about the new solution count. Hence up to d-2 propagations might be needed. Even in the case that the deletion results in no invalidations of solution extending from Parent(N), the complexity is the same.

Case(ii) Subtree extends descendant solutions. If the subtree has a depth of S_d and a branching factor S_b, then with a complexity of $S_d * S_b$ we can determine at N the count of number of complete descendant solutions that had been extended by subtree(N). In reality, this number will be lesser as a probe is likely to be encountered before reaching the leaf nodes on every path from N. A further effort of at-most (d-2) would be required to calculate the new counts and propagate them towards the root of the document.

3.7 The PathMatch algorithm

In the earlier scheme NodeMatch, we saw that only those nodes that participated in a solution stored the pre-computations. Another alternative is for all nodes in the path of a solution to store the same pre-computed information. As a large portion of the logic is essentially the same we do not repeat it here. Instead we just refer to functions mentioned in NodeMatch and provide extensions where applicable.

The Initial pre-computation phase of PathMatch

The initial pre-computation phase is not very different from that of NodeMatch's *find_pattern* function (Figure 3.8).

The only difference is in case '4' of the *find_pattern* function, i.e, given a probe that does not match any node. The NodeMatch model does not mark the probe to be stored but marks it to be forwarded. If we are going to store information at intermediate nodes too, we will mark any probe that is forwarded as stored, provided it's descendants match some extension. As an alternative, we can do without the *mark_as_stored* flag and instead, when the probe returns, probes that have not been extended can be deleted.

One of the interesting situations is when there are descendant node matches and the complete sub-tree of that descendant is matched. We propagate the count of the number of such descendants back to the parent/ancestor using a set of counts (desc_position_count), we need to store this at all the intermediate nodes as well. We are only interested in the counts of complete subtree solutions for each descendant at the intermediate nodes, this would suffice as the intermediate node will anyways not participate in parent child relationship. The implication of this is that this intermediate node is of interest only to its parent/ancestor node to which some new subtree that extends its next_position_child is added/deleted.

To summarize, the changes to the NodeMatch algorithm for it to store pre-computed

```
1:  Function check_for_extension_new(Probe P_n, Query Node Q_x, Flag childFlag)
2:  Set flag extends_soln to false
3:  for Each RP_i IN RetProbe[1..n]→probe do
4:     {RetProbes[i] or RP_i refers to the i^th probe of the i^th child of N}
5:     if Q_x matched in RP_i then
6:        {This probe has been extended by RP_i}
7:        Set extends_soln to true
8:        if number of matched nodes at Subtree(Q_x) in RP_i > Matched nodes in Subtree at Q_x of P_n then
9:           Copy all matched nodes of the sub-tree(Q_x) of RP_i into P_n
10:       end if
11:       if Subtree(Q_x) in RP_i is complete then
12:          if childFlag == true then
13:             Increment next_child_counts[Q_x] by 1
14:          else
15:             Increment next_desc_counts[Q_x] by RP_i →total_solution_count
16:          end if
17:       end if
18:    end if
19: end for
20: if extends_soln == true then
21:    Mark P_n to be stored
22: end if
23: End of function check_for_extension_new
```

Figure 3.21: Function check_for_extension_new()

information at intermediate nodes is essentially one step. In the function *check_for_extension* (Figure 3.14) we use a flag '*extends_soln*'. If the solution is extended by any of the nodes children, it is set to true and this probe is marked to be stored. Once it is marked to be stored, the *compute_counts_and_merge* function (Figure 3.15) ensures that the pre-computations are stored at the intermediate nodes. The modified algorithm for *check_for_extension* (Figure 3.14) is given below in Figure 3.21.

3.8 Incremental maintenance of PathMatch

As discussed in NodeMatch, we shall cover the insertion and deletion of entire subtrees on the existing document D. The essential operations involved in incrementally maintaining the pre-computed values remain the same.

```
1:  Function check_ancestor_exists()(Probe insert_Probe, Node N
2:  if insert_probe has complete solution then
3:     RETURN true {A path must exist to the root}
4:  end if
5:  Make a list of descendant query positions desc_list that have counts > 0
6:  Set starting node St_n as parent(N)
7:  while St_n is not the root of D do
8:     if node matches any of the desc_list then
9:        RETURN true
10:    end if
11:    St_n = Parent(St_n)
12: end while
13: RETURN false
14: End of function check_ancestor_exists
```

Figure 3.22: Function check_ancestor_exists()

3.8.1 Insertion of a complete sub-tree using PathMatch

As in NodeMatch, we use the *find_pattern* method (Figure 3.8) to pre-compute information in the new sub-tree being added. However, we need to make a minor modification. For PathMatch, the *find_pattern* method stores intermediate probes only if the information it contains is used by some ancestor node. In this case, if we have a intermediate node that contains a descendant match without any ancestor present in the new subtree, before deleting it we need ascertain that no ancestor exists in the main document along the path of the insert operation. To support this, the *find_pattern* method is passed an additional pointer that points to the parent node at which the new subtree is being inserted. The remaining operations are the same. After the *insert_probe* is computed, we use the method of *check_ancestor_exists* (Figure 3.22) to determine if intermediate nodes are required to be created. If it returns true, while using the *correct_parent_increment* function (Figure 3.17), all the nodes till an ancestor/another intermediate is encountered will store a copy of the *insert_probe*.

1: **Function delete_subtree(Node N)**
2: **if** $N \rightarrow ProbeisNULL$ **then**
3:     RETURN
4: **end if**
5: Set delete_probe as copy of $N \rightarrow Probe$
6: Set up_forward to false
7: **if** N matches a next_position_child Q_c of Parent(N) **then**
8:     **if** Probe at N matches complete subtree(Q_c) **then**
9:         Set up_forward to true
10:     **end if**
11: **end if**
12: Call Parent(N).correct_parent_decrement(delete_probe, up_forward, N)
13: delete intermediate probes that are not used anymore.
14: **End of function *delete_subtree***

Figure 3.23: Function delete_subtree() for PathMatch

3.8.2 Deletion of a complete sub-tree using PathMatch

As in NodeMatch, The *delete_subtree* method (*Figure 3.23*) receives a Node N as input, It needs to delete all the nodes in its sub-tree and also update all the probes in its parent and ancestors that have solutions extended by the sub-tree of N. The key difference in this method is that, unlike NodeMatch, we do not have to search the sub-tree being deleted to create the *delete_probe*. If the node whose sub-tree is being deleted does not contain a probe, it implies that no solutions extend using its sub-tree. Thus, no recalculations needs to be done and only the actual deletion of the nodes are required. If it contains a probe, that probe is itself the *delete_probe*. Like, in the case of NodeMatch, we determine if it is an extension of a parent-child solution, in which case we use the flag *up_forward*. The other operations performed are the decrements of descendant counts and the recalculation of the total_solution_counts. Additionally, due to the sub-tree being deleted, probes at the intermediate nodes along the path to the root of the document may be rendered useless, these are deleted. When the delete_probe reaches the root, we can get the total count of the number of solutions that have reduced due to the delete operation. This is deducted from the total number of solutions to the query.

3.8.3 Complexity analysis of PathMatch algorithm

The pre-computation phase:

The pre-computation phase of PathMatch is not very different from that of NodeMatch. Both the methods use the *find_pattern* method. The additional complexity in the case of PathMatch arises from having to determine if an intermediate node is required to store a probe. The cost of each of these checks is O(d) where 'd' is the depth of the intermediate node. Looking at complexity in terms of storage, we are storing a larger number of probes compared to NodeMatch. The count of extra probes can be determined by the number of times the check on whether an intermediate probe needs to be stored returns true.

The insert operation

The insert operation in the case that a single node is being added, we only need to check the node is an extension of its parent's solution. In this case, 'd' node operations that re-compute the total_solution_count is required. If an entire sub-tree is being added, then the complexity arises in executing the *find_pattern* method in the new sub-tree. In addition, We also need to determine the intermediate nodes that are to be stored in the new subtree, and the new intermediate nodes that are to be created in the existing document tree. This complexity is O(d), where d is the depth of the intermediate node after the sub-tree has been inserted to the document.

The delete operation

The complexity of the deletion of a node or entire sub-tree with or without descendant matches are now the same. The total cost involved is equal to the cost for recomputing the total_solution_count at each node from the parent node of the delete operation till the root of the document. If we encountered an intermediate node that has no positive

counts for any of the descendant position, it is deleted. However, if a descendant position count becomes zero, we need to ascertain that there is some ancestor of this node that uses one of the other positive descendant match counts. The cost of this equates to 'd' node operations where 'd' is the depth of the intermediate node.

Chapter 4

Experimental study

In this chapter, we discuss the various experiments that we have used to identify the different trade-offs between the two schemes and the different performance factors. We try to bring forth all the decision parameters that need to be considered before deciding on using one of the two schemes. We outline the different data-sets, queries and update operations in section 4.1. The results obtained for the various experiments are discussed in section 4.2. Finally, we summarize our findings in section 4.3.

4.1 Experimental setup

We implemented both the algorithms NodeMatch and PathMatch in its entirety using the C++ programming language. The experiments were carried out on an Intel 1.3GHz Centrino processor with 256 MB main memory. The machine runs on the Redhat 9.0 operating system. The code has been compiled using the 3.2.2 version of the gcc compiler. The experiments are designed to highlight the applicability and efficiency of the pre-computation based methods, additionally we try to identify the tradeoffs between the two pre-computation algorithms. The amount of pre-computation stored is highly dependent on the nature of the database and the queries. Thus, to be fair

to both the algorithms, we perform tests on three different data-sets that vary in their properties of fan-out, depth, recursion of elements but has the same number of elements. Additionally, we run a large mix of queries against them. We also include the results that illustrate the scalability of the schemes over larger data-sets.

4.1.1 The data-sets

The data-sets used for the experiments are all synthetic data-sets that have been generated by using the Niagara XML Data Generator [1]. The template configuration file used for these data-sets is provided in the Appendix A[1]. Here, we provide a broad idea of the general nature of each data-set and the influence of their properties on the two algorithms. The data-sets are summarized in Table 4.1.

Data-set1: Low fan-out, large depth.

This data-set will test the capacity to handle descendant matches and the influence the large depth has on the number of intermediate nodes stored. We shall also see how the incremental maintenance during updates varies for both the algorithms. We use a depth equal to 8, with an average fan-out of 4.

Data-set2: Large fan-out, low depth.

In this data-set we use an average fan-out of 27, with a depth of 4. This data-set brings out the lesser role that intermediate nodes have to play in case of documents with low depth. It also restricts the kind of ancestor-descendant queries that can be matched.

Data-set3: Average depth and fan-out.

This data-set forms an intermediate type of data-set in comparison to the above two. It uses a height of 5 with an average fan-out of 13. This neutral data-set is used to obtain the average results of the performance of the two algorithms.

Data-set4: Data-set with 100,000 elements.

[1] Adapted from the Niagara XML Data Generator package

Data-sets			
$Data - set$	# of Elements	Depth	Avg. Fan-out
1	21875	8	4
2	21875	4	27
3	21875	5	13
4	100000	4	75
5	300000	4	100

Table 4.1: The data-sets used in the experiments.

This data-set has a depth of 5 and an average fan-out of 75. This data-set is primarily used to illustrate the ability of NodeMatch and PathMatch to handle large data-sets.

Data-set5: Data-set with 300,000 elements.

This data-set has a depth of 5 and an average fan-out of 100, This data-set too like data-set 4 is used for experiments that illustrate the scalability of both the schemes to handle large data-sets.

All the above data-sets include recursion of elements and semi-structured properties. The data-sets 1, 2 and 3 operate on an average of 20000 elements.

4.1.2 The boolean twig queries and update operations

This section details the various types of queries that are executed against the three variations of the data-set. We use the labels of P, C, A, D and N to represent Parent, Child, Ancestor, Descendant and N respectively . A $P \rightarrow C$ implies that C is a child of P, similarly, $A \rightarrow D$ is used to express A is an ancestor of D. $A \rightarrow DC$ means that A is an ancestor of D and has a child C. We first generate queries that involve only parent-child relationships. This set of queries forms 'Query set1' as shown in Table 4.2. The second set of queries 'Query set2' involve both parent-child and ancestor-descendant relationships to be matched. Query set2 is shown in Table 4.3. Lastly, we create a set of update operations (Query set3) that are to be performed on a previously pre-computed document. Query set3 is shown in Table 4.4. For 'Query set3', we shall use a standard query to perform the initial pre-computation, This standard query has both ancestor-

Query set1	
Query	Description
Q1	Small twigs with a node and two children ($P \rightarrow CC$)
Q2	Twigs with a large number of children ($P \rightarrow CCCCCC$)
Q3	Twigs with large depth ($P \rightarrow P \rightarrow P \rightarrow P \rightarrow P \rightarrow C$)
Q4	Bushy queries
Q5	Queries that have a partial match
Q6	Queries with no matching solution

Table 4.2: Query set1 containing only parent-child matches.

Query set2	
Query	Description
Q7	Single descendant twig ($A \rightarrow D$)
Q8	Twig with one descendant and one child ($A \rightarrow DC$)
Q9	Twig with multiple descendant matches ($A \rightarrow DDDCC$)
Q10	multilevel descendant matches ($A \rightarrow D \rightarrow D \rightarrow D$)
Q11	Parent, child and descendant to match, with a sub-tree at descendant
Q12	Multiple descendant sub-trees to match
Q13	Descendant match at deep nested level
Q14	Queries that have a partial match
Q15	Queries with no matching solution

Table 4.3: Query set2, queries with descendant matches.

descendant and parent-child relationships to match. We gather results such as time taken to perform pre-computations and the time-taken on answering repetitive queries. Other time measurements include the update operations of insert and deletes. We also measure the amount of memory used to store the pre-computations.

Query set3	
Query	Description
I1	Insert of a single node that participates in a solution
I2	Insert of a single node that does not participate in a solution
I3	Insert of a entire sub-tree that has no solutions
I4	Insert of a entire sub-tree that contains complete and partial solutions
D1	Deletion of a single node that participates in a solution
D2	Deletion of a single node that does not participate in a solution.
D3	Deletion of a entire sub-tree that has no solutions
D4	Deletion of a entire sub-tree that contains complete and partial solutions

Table 4.4: Query set3, Operations that test incremental maintenance of updates.

4.2 Experiments and Results

We performed the following set of experiments using the various data-sets and queries mentioned in the experimental setup section. For each of these experiments discussed below, we obtained the results from runs of both the algorithms.

Pre-computation times for various queries: We ran the queries Q1-Q15 listed in Tables 4.2 and 4.3 on data-set 1, data-set 2 and data-set 3 (Table 4.1).

Update performance: We ran the various insert and delete operations listed in Table 4.4 on data-set 1, data-set 2 and data-set 3 (Table 4.1). For the delete operation, we obtain an additional result that splits the time taken to perform the delete into two components, update time and validation time. Update time is the time taken to perform the actual updates of the pre-computations. Validation Time is the time taken to compute only the result of the operation.

Comparison of space requirements: The memory used in NodeMatch and PathMatch directly corresponds to the number of probes stored in each case. The number of probes stored depend directly on the number of nodes that match a query. In this experiment we vary the percentage of nodes that contain repeated element tags.

Experiments that determine the effect of varying fan-out on updates: Here we vary the fan-out of the data-set while holding the depth constant.

Experiments that determine the effect of varying depth on updates:. Here we vary the depth of the data-set while holding the fan-out constant.

Scalability comparison: Queries and update operations are executed on data-set 3, data-set 4 and data-set 5 (Table 4.1) to illustrate the times taken for these operations on large data-sets.

4.2.1 Performance on various queries

As mentioned in the experimental setup, we ran NodeMatch and PathMatch on the three different data-sets using queries Q1-Q15 described in Tables 4.2 and 4.3.

Pre-computation times for parent-child queries:

Figures 4.1, 4.2 and 4.3 show the run-times obtained for the queries Q1-Q6 that involve only parent-child relationships on data-sets 1, 2 and 3 respectively. As expected, PathMatch takes more time than NodeMatch to evaluate most of the queries. This is attributed to the following reason, PathMatch tries to determine whether the probes being processed at nodes that do not match the query are to be stored as intermediate probes.

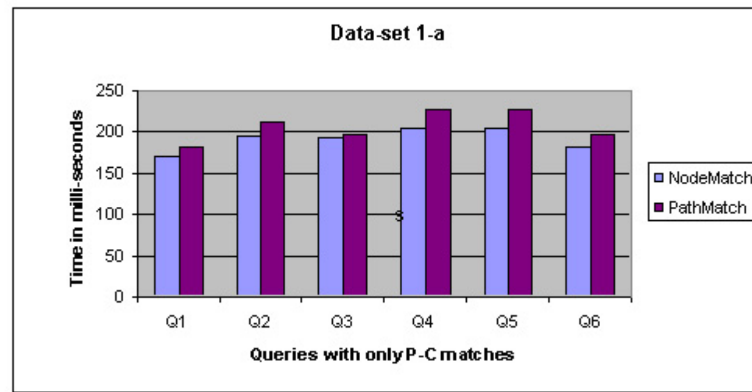

Figure 4.1: Pre-computations for Data-set1 on Queries Q1-Q6

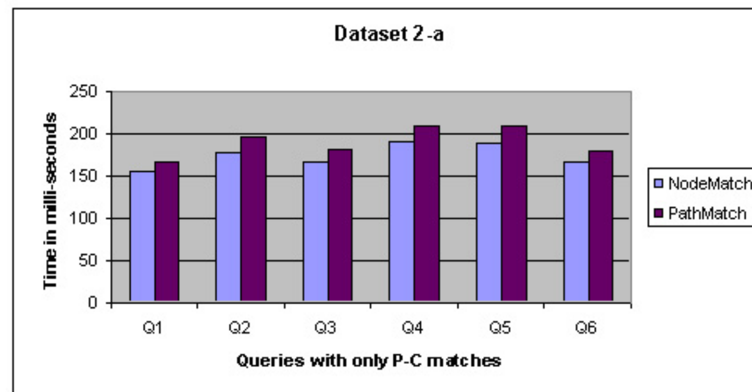

Figure 4.2: Pre-computations for Data-set2 on Queries Q1-Q6

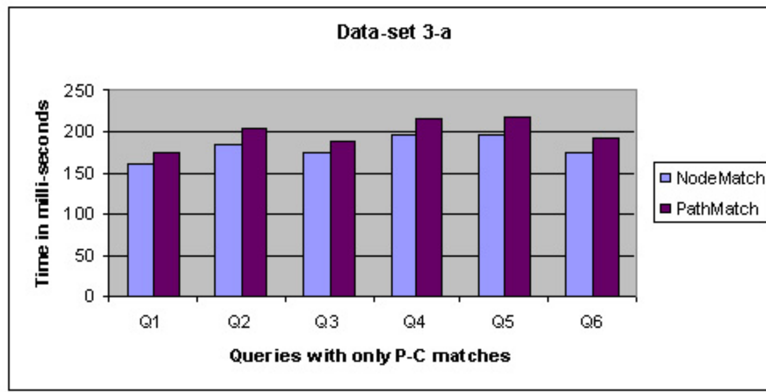

Figure 4.3: Pre-computations for Data-set3 on Queries Q1-Q6

Pre-computation times for queries including ancestor-descendant matches:

The pre-computation times for the queries Q7-Q15 involving ancestor-descendant matches are shown in the Figures 4.4, 4.5 and 4.6. Here too we observe the same delay in the PathMatch algorithm. We also notice that, as the number of elements of the document processed is the same, there is not much of a difference across the pre-computation times of the different queries. Any difference can be contributed to larger query twigs leading longer iterations during processing of each node. Also, it can be noticed that queries with a larger number of descendant positions like Q9 and Q12 take longer times than the others.

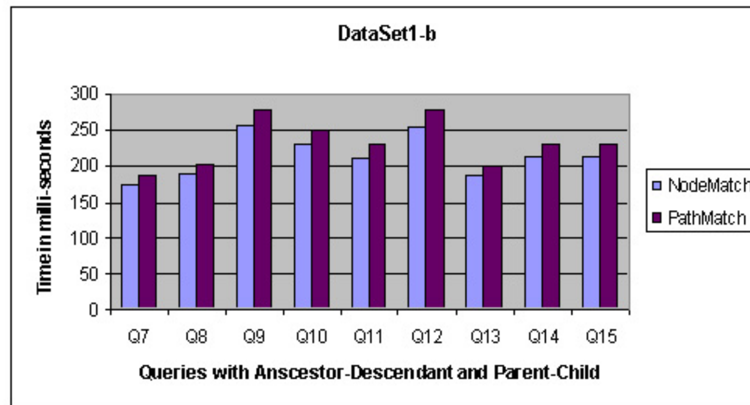

Figure 4.4: Pre-computations for Data-set1 on Queries Q7-Q15

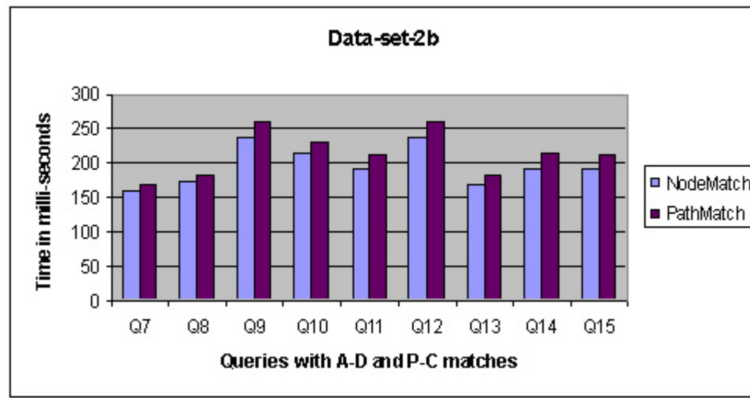

Figure 4.5: Pre-computations for Data-set2 on Queries Q7-Q15

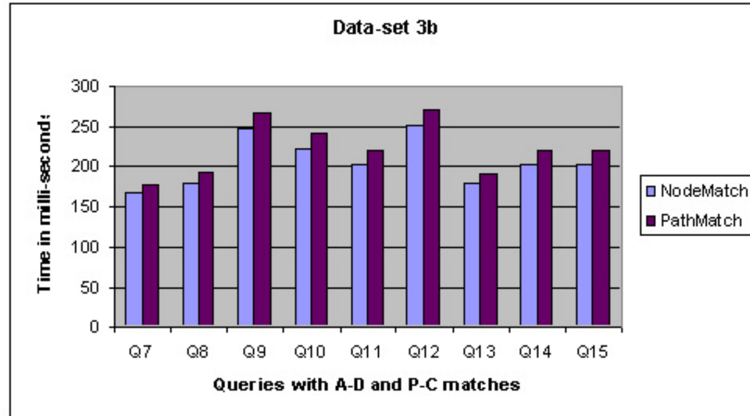

Figure 4.6: Pre-computations for Data-set3 on Queries Q7-Q15

4.2.2 Update Performance

We pre-computed the result for Q11 on data-set1, data-set2 and data-set3(Table 4.1). We performed the various update operations on this pre-computed data. If there is no pre-computations stored for a given query then delete operations do not affect the existing state of the document. Inserts can change the state of the document only if the new data being added contains matches to the query.

Deletion of nodes and sub-trees

The time taken to update the pre-computations stored upon deletion is shown in Figures 4.7, 4.8 and 4.9. The graphs corresponding to the three data-sets data-set1, data-set2 and data-set3(Table 4.1) illustrate that there is not much difference between computa-

tion times required by the two algorithms for the case that a single node that matches the query is deleted at the leaf node. This is reasonable as both the algorithms essentially perform the same operations that correspond to updating the nodes along the path to the root. For the cases where the root of the sub-tree being deleted does not match the query, we see that PathMatch is distinctly faster as no changes to any pre-computations of the ancestors are required. NodeMatch, on the other-hand has to ensure that no solutions exist in the sub-tree being deleted and hence is considerably slower. For such cases, we find that NodeMatch is slower by more than an order of 10 times as compared to PathMatch. If the root of the sub-tree being deleted matches the query, NodeMatch is comparable in performance to PathMatch. In this case, the re-computations are done at the ancestors and parent nodes for the NodeMatch scheme and additionally at the intermediate nodes for the PathMatch scheme.

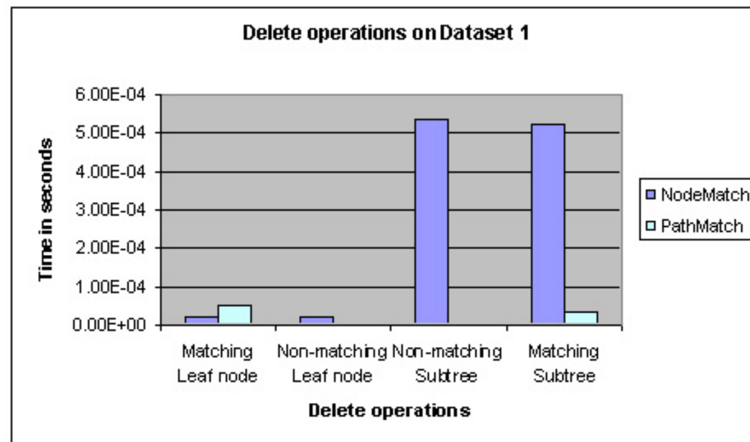

Figure 4.7: Delete operations on Data-set1

4.2.3 Validation Time

This describes how the times observed for the delete operations in Figures 4.7-4.9 can be seen as two components, one that determines if a solution still exists after a delete operation and two, the actual updates of the pre-computations stored. In this section we show the running times taken to answer the query asking if a solution exists upon

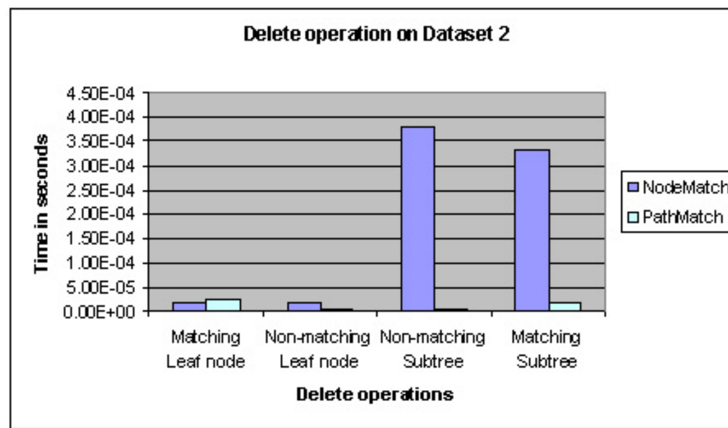

Figure 4.8: Delete operations on Data-set2

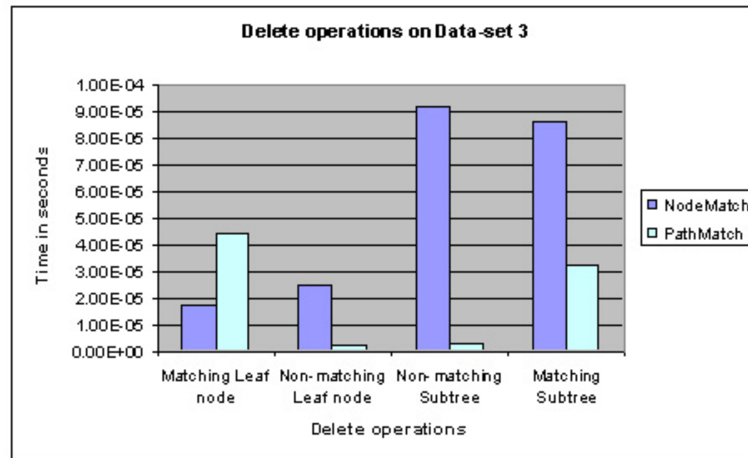

Figure 4.9: Delete operations on Data-set3

the occurrence of a delete operation. Here we are assuming that the updates on the pre-computations are deferred. i.e. we just need to determine whether the solution still exists after the proposed delete operation is performed and we do not update the pre-computations stored. As shown in Figure 4.10, PathMatch is faster than NodeMatch. We note that NodeMatch spends time searching a sub-tree that contains no solutions, where as PathMatch knows this information by just checking if the root node of the delete operation contains a pre-computation probe. We need to take note that we are only considering cases in which the root node of the sub-tree being deleted is not a matching child node of the probe in its parent. If the root node matches the a child position of its parents probe, the performance of PathMatch will reduce to that of the

regular delete operation as the entire re-computations will be required to determine if a solution exists.

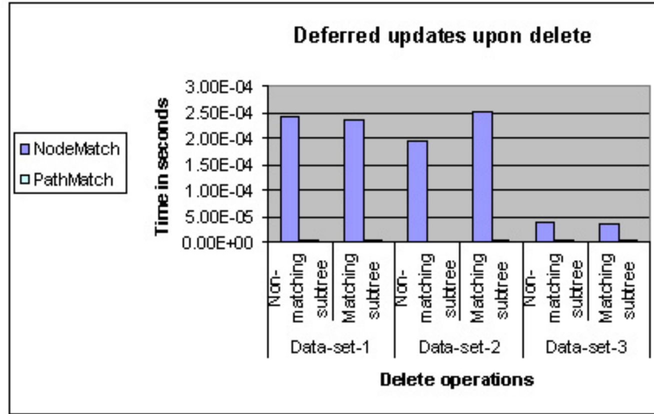

Figure 4.10: Validation time for delete operations.

Insertion of nodes and sub-trees

Similar to delete, we ran the insert operations described in table 4.4 on the three data-sets described in Table 4.1 and the results are shown in Figures 4.11, 4.12 and 4.13. The insert operations take nearly the same time for both NodeMatch and PathMatch as both the methods essentially update the same set of nodes along the path to the root. PathMatch takes slightly longer as intermediate probes too have to be updated.

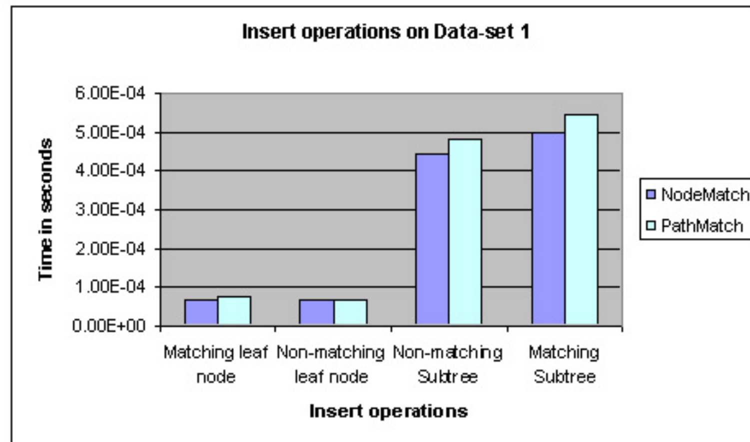

Figure 4.11: Insert operations on Data-set1

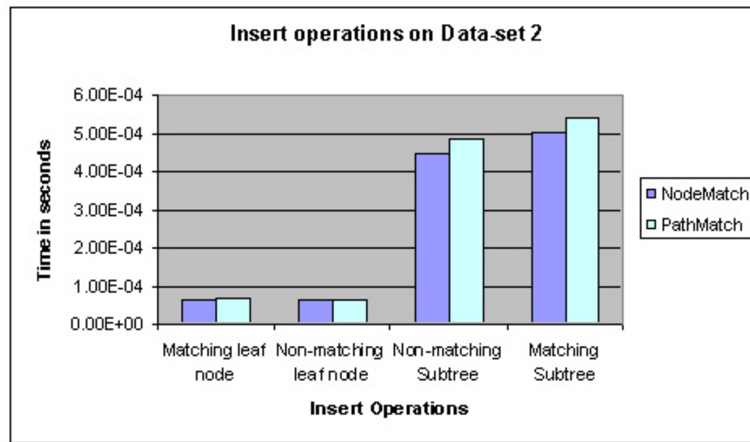

Figure 4.12: Insert operations on Data-set2

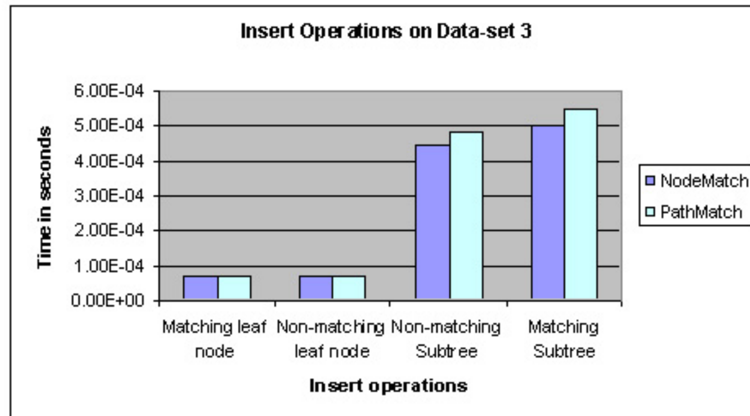

Figure 4.13: Insert operations on Data-set3

4.2.4 Comparison of Space Requirements

To get an estimate of the amount of memory required to store the pre-computations we vary the percentage of elements that have repeated tags. This leads to an increased number of element nodes matching the query, hence more pre-computed elements. We have used query Q11 to get a count of the number of probes stored. We measured the number of probes stored for various number of repeated elements in a data-set of 20000 elements. The results obtained by varying the percentage of repeated element tags between 0.05% to 0.4 % is shown in Figure 4.14. Generally, the PathMatch algorithm will store nearly twice as many probes as NodeMatch, This is because all the nodes between a pair of ancestor-descendant matches in the query store the probes. Additionally along

each path from root to leaf, all the nodes between a match and the root of the document also store probes.

To the define two extremes in the number of probes stored, we consider two cases. Case(i) All the leaf nodes of a document matches a descendant position of a query. In this case, all the nodes in the tree would store pre-computed probes for PathMatch. In case of NodeMatch, apart from the root, only the leaf nodes would store the probes. Case(ii) There are no descendants to match in query and the query has its root matching the root of the document or the root's immediate children. In this case, both NodeMatch and PathMatch store the same number of probes.

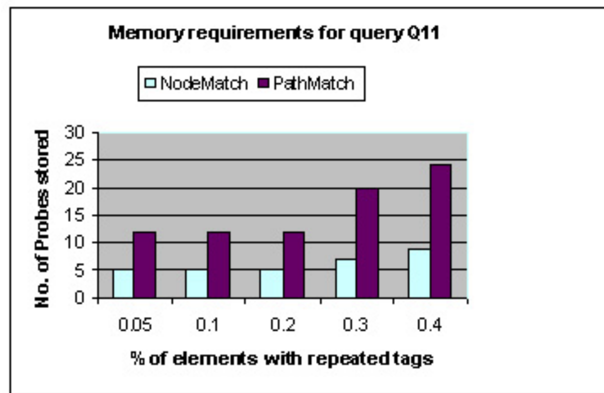

Figure 4.14: Memory requirements for increased repetition of element tags

4.2.5 Update times for varying Fan-out with constant Depth

In this section, we determine the influence of the fan-out of the data-set on the time required to perform the update operations. We used a data-set with a depth of 4 and varied the average fan-out. We used fan-outs of 6, 12, 22 and 42, these resulted in documents that contained 262, 1575, 9700, 70515 element nodes respectively. We obtain the times for the delete and insert operations as shown in Figures 4.15 and 4.16 respectively. As seen from Figures 4.15, the delete times increases considerably with increase in fan-out. For the data-set with fan-out 6, the root of the sub-tree being deleted contained a probe, hence both the algorithms executed with similar execution

times. Looking at the insert operations (Figure 4.16), we see that there is a little or no increase in insert times with increasing fan-out. This is because during inserts only parent nodes and ancestor nodes are accessed.

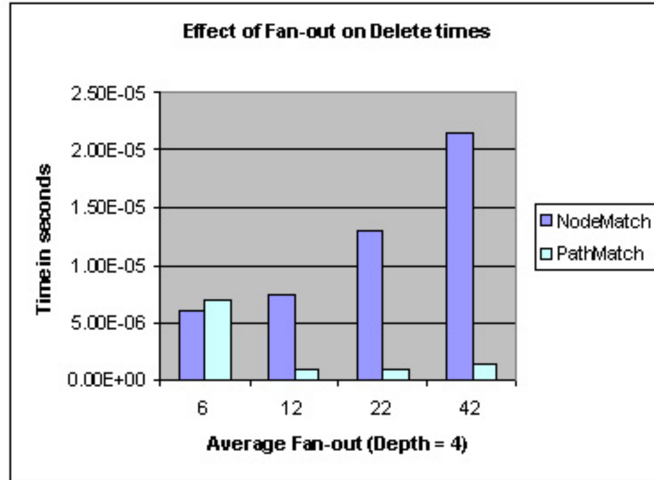

Figure 4.15: Effect of varying the fan-out on delete operations

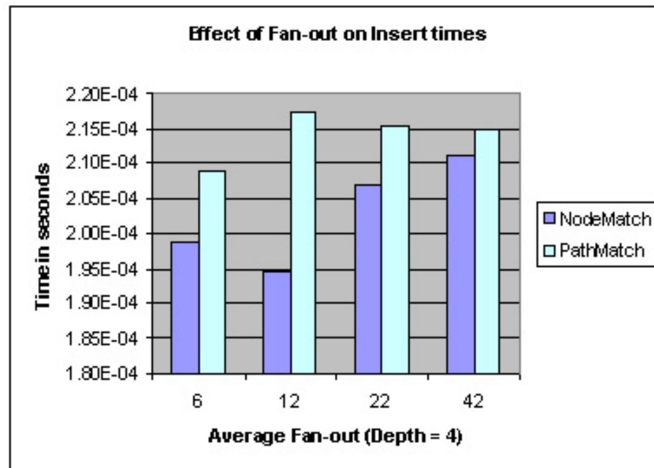

Figure 4.16: Effect of varying the fan-out on insert operations

4.2.6 Update times for varying depth with constant fan-out

To determine the influence of depth of the XML document on the time required for the update operations. We used a data-set of with an average fan-out of 6 and varied the depth. We used depth values of 4, 5, 6 and 7, these resulted in documents that contained 262, 1550, 10570, 56690 element nodes respectively. We obtain the times for the delete

and insert operations as shown in Figures 4.17 and 4.18. We observe that for documents with increasing depth the delete operations take longer. We also see that there is a very sharp increase for NodeMatch as the depth increases. This is as expected. From the level at which the updates take place, with increased depth there are a much larger number of paths that NodeMatch needs to check to determine solutions existing in the sub-tree being deleted. For the insert operations we see that, both NodeMatch and PathMatch take nearly the same time, with increasing depth we see a gradual increase in the time taken for the updates.

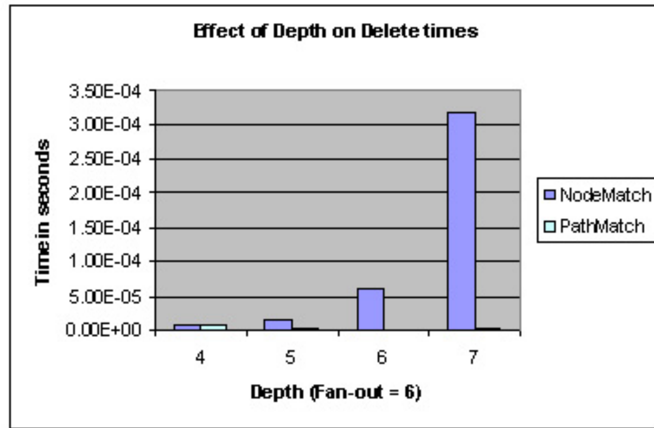

Figure 4.17: Effect of varying the depth on delete operations

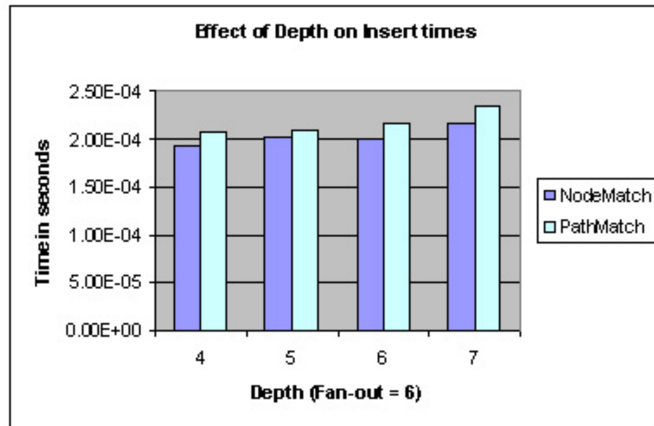

Figure 4.18: Effect of varying the depth on insert operations

4.2.7 Scalability Comparison

We test the scalability of the two algorithms in terms of time taken for performing the pre-computations, insert of sub-trees and deletes of sub-trees. We vary the number of elements in the XML document. Here we use data-set 3, data-set 4 and data-set 5 (Table 4.1). They contain around 20000, 100000, 300000 elements respectively. The results obtained have been summarized in the graphs shown in Figures 4.19, 4.20 and 4.21.

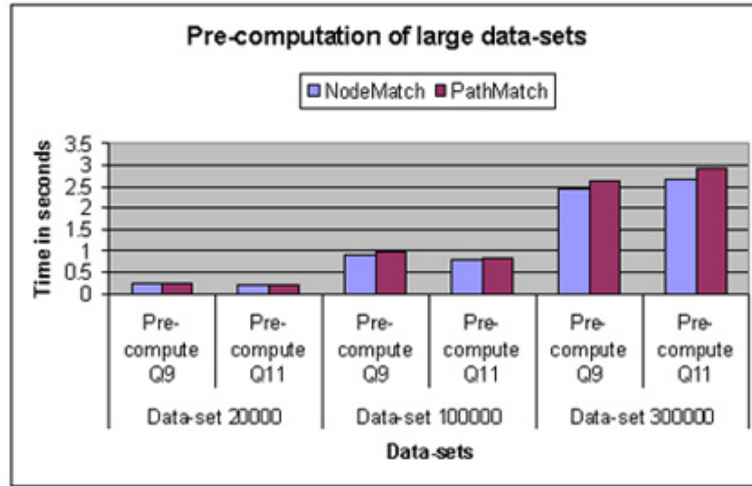

Figure 4.19: Pre-computation on large data-sets

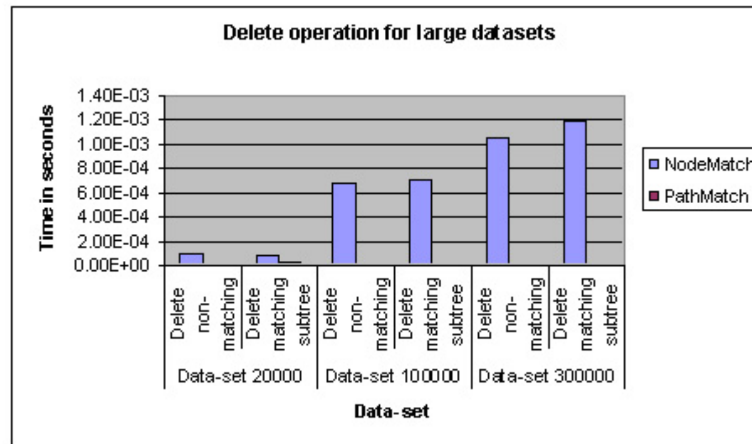

Figure 4.20: Delete operations on large data-sets

The value for the pre-computation of 20000 data-set is nearly a fifth of that of the 100000 data-set which in turn takes nearly a third of the time taken for the 300000 data-

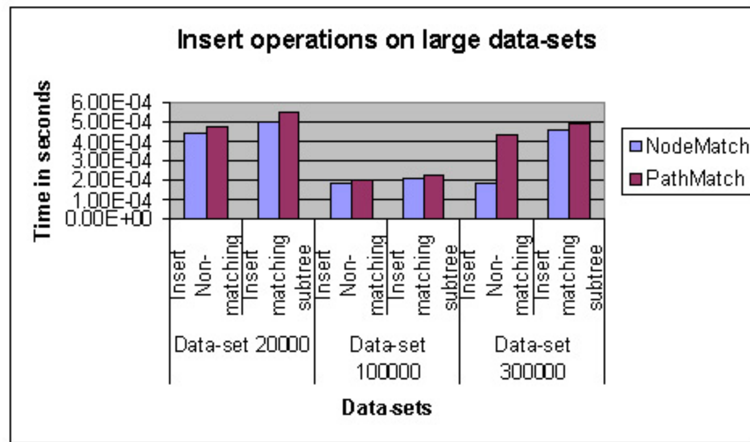

Figure 4.21: Insert operations on large data-sets

sets. This shows that the complexity of the pre-computation phase in proportional to the number of elements in the database. Figure 4.21 shows the performance on inserts, we see that even on large data-sets the time taken for both the algorithms is nearly constant and is affected only by the depth of the tree at which the insert operation is performed.

4.3 Summary

We summarize the trade offs and strengths of NodeMatch and PathMatch using Table 4.5. In the table, we describe the various factors that affect the performance of both the algorithms. We also briefly summarize the essential differences between NodeMatch and PathMatch in executing operations such pre-computations, inserts and deletes. From all the experimental results we have gathered we find that NodeMatch is faster than PathMatch in pre-computation phase and for insert operations. However, for both pre-computations and inserts, NodeMatch and PathMatch differ only by an order of few milli-seconds (around 10-20). In contrast, PathMatch is generally faster than Node-Match for delete operations by more than an order of 10 times. PathMatch also has the advantage that for delete operations we can choose to validate the existence of the solution to a query before actually updating the pre-computations. PathMatch has the

clear advantage that all the operations are bottom-up, that is from leaf to the root. This avoids costly sub-tree searching operations that affects NodeMatch. Thus, given enough memory, PathMatch can provide excellent performance for dynamic XML databases.

NodeMatch Vs PathMatch		
Factor/operation	NodeMatch	PathMatch
Pre-computation	Faster, lesser memory needed	Slower more memory needed
Insert operation	Update only matching parent, ancestors nodes. If no matches present then same time requirements for both.	Update matching parent, ancestor and intermediate nodes.
Delete operation without matches	Slow, requires searching in sub-tree being deleted.	Very fast, as no probe is stored at the root of sub-tree being deleted, nothing is done.
Delete operation with matches	Slow, searching sub-tree required. Comparable to PathMatch only if root of subtree being deleted contains a probe.	Fast, as root of sub-tree contains a probe that is used to update the ancestors till the root.
Larger Data-set	Constant increase as PathMatch in pre-computations and inserts. Suffers more in deletes.	More number of probes to be stored assuming more matches.
Increased Depth	Suffers during deletes, as a delete at a level nearer to the root involves a larger sub-tree to be searched.	More ancestors are checked to determine if a probe is to be stored. Larger number of intermediate nodes stored
Increased Fan-out	Pre-computation takes longer, increases same rate as PathMatch. No effect on insert. Deletes take longer for same reason as increased depth	Pre-computations take longer. No effect on delete or insert
Repetition of labels	Number of additional probes stored equals the number of repeated labels matching the query	Increased number of probes being stored as intermediates are stored across a larger number of paths.
Deferred delete	Can answer query only after searching the sub-tree being deleted	Can answer query immediately if the root of the sub-tree does not match a child position of the probe of its parent.
Trace solutions to access actual nodes of a solution	Theoretically possible, required searches in sub-trees with matching nodes.	Possible, as there are paths to all solutions from the probe at the root of the document.

Table 4.5: The comparison of NodeMatch and PathMatch

Chapter 5

Conclusion

In this thesis, we have presented two algorithms that uses pre-computations to answer frequent queries against dynamic XML databases. We focused on computing solutions to boolean twig query patterns. For boolean queries, we are only required to determine the existence of a given twig query in an XML document and do not have to retrieve the actual data nodes that match the query. We also compute the number of solutions that are present in the document against which the query is executed.

As with any pre-computed information it is subjected to the curse of updates. We use a methodology of incremental maintenance in order to maintain the correctness of the pre-computed information upon updates to the document on which the pre-computations are built. The challenges we have faced in this task include determining what pre-computed information is to be stored and where to store it. Another critical challenge was to limit the number of nodes that need to be accessed in order to update the pre-computations when the XML document is subjected to updates.

We designed a data structure called the '*probe*' that contains enough information to determine the extent of the pattern match of a twig query that is present in the entire sub-tree of any given node of the XML document. The first algorithm presented called NodeMatch stored pre-computation probes only at the nodes that matched the query.

In the second algorithm named PathMatch we store the pre-computation probes along all the intermediate nodes between ancestor-descendant matches of a twig query. Probes are also stored along the root of the document to every complete twig query match that exists.

With the development of PathMatch, we have a pre-computation based algorithm that only accesses nodes in the leaf to root manner. Thus the complexity of update operations are governed by the height of the XML document tree. The NodeMatch algorithm on the other-hand is seen to be faster for the pre-computations phase and for inserts. Thus, if the document is not going to be subject to many delete operations it might be feasible to use NodeMatch instead of PathMatch.

Future direction:

The current research can be extended to include support for recursive query elements and ordering. There are various applications in which pre-computations of queries can be efficient. A few possible directions include the publisher-subscriber system and XML document access control. Using PathMatch, we can trace the path from the root to the nodes that contain the solution. Thus the complete records of matching solutions can be retrieved. We also observed that for the purpose of boolean twig queries, NodeMatch is equally competitive and suffers only from having to search the sub-tree upon deletes. We could extend NodeMatch to include additional information at each probe stored at a node to represent its sub-tree. Future work can also include compressing the information stored and the re-use of pre-computations stored in one query to solve another query.

We conclude this thesis by emphasizing that pre-computations are an effective way to provide results to repetitive queries and that incremental maintenance provides the efficiency required upon the updates for pre-computation based query processing to be a reality.

Appendix A

Niagara XML Data Generator

A.1 Configuration file template

Prefix of filename for generated documents

Random number generator seed

Number of documents to generate

Number of levels in the path tree (n)
Minimum fan-out of level 0 of path tree (root) M̂aximum fan-out of level 0
Minimum fan-out of level 1 M̂aximum fan-out of level 1
...
Minimum fan-out of level (n-2) M̂aximum fan-out of level (n-2) {Level (n-1) is the leaf level}

Fraction of internal path tree nodes with direct recursion in the tag name.
Fraction of internal path tree nodes with indirect recursion in the tag name.
Fraction of internal path tree nodes with repetition in the tag name.
Fraction of leaf path tree nodes with repetition in the tag name.
Fraction of path tree nodes with repetition in the tag name(internal or leaf).

Total number of XML elements to generate (per document)
Zipf value (skew) of element frequency distribution
Assignment of element frequencies to path tree nodes (asc/desc/rand)
Spread of element frequency distribution around average (non-determinism).

Number of distinct text words to generate
Total number of text words to generate (over all documents)
Zipf value (skew) of text word frequency distribution
Probability of an internal node element of the path tree having text word(s)
Probability of an leaf node element of the path tree having text word(s)
Maximum number of text words per element

Bibliography

[1] Niagara XML Data Generator(2001), http://www.cs.wisc.edu/niagara.

[2] XML Schema W3C recommendation(2001), http://www.w3.org/tr/xmlschema-2.

[3] W3C(1999) XML path language (XPath) 1.0, http://www.w3.org/tr/xpath.

[4] W3C(2000) Extensible markup language(XML)-1.0,2nd. edn, http://www.w3.org/TR/REC-xml/.

[5] W3C(2003) XQuery 1.0:An XML Query language, http://www.w3.org/TR/xquery/.

[6] World Wide Web Consortium (W3C), http://www.w3.org.

[7] Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, page 141. IEEE Computer Society, 2002.

[8] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 38–49. Morgan Kaufmann Publishers Inc., 1998.

[9] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. *SIGMOD Rec.*, 30(2):497–508, 2001.

[10] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient Incremental Validation of XML Documents. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 671. IEEE Computer Society, 2004.

[11] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321. ACM Press, 2002.

[12] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML documents with XPath Expressions. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, page 235. IEEE Computer Society, 2002.

[13] Li Chen, Elke A. Rundensteiner, and Song Wang. XCache: a semantic caching system for XML queries. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 618–618. ACM Press, 2002.

[14] Li Chen, Song Wang, Elizabeth Cash, Burke Ryder, Ian Hobbs, and Elke A. Rundensteiner. A fine-grained replacement strategy for XML query cache. In *WIDM '02: Proceedings of the 4th international workshop on Web information and data management*, pages 76–83. ACM Press, 2002.

[15] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 134–144. ACM Press, 2003.

[16] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting Twig Matches in a Tree. In *Pro-*

ceedings of the 17th International Conference on Data Engineering, pages 595–604. IEEE Computer Society, 2001.

[17] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *VLDB*, pages 263–274, 2002.

[18] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *The VLDB Journal*, 11(4):292–314, 2002.

[19] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[20] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[21] Hyunchul Kang, Hosang Sung, and ChanHo Moon. Deferred incremental refresh of XML materialized views: algorithms and performance evaluation. In *CRPITS'17: Proceedings of the Fourteenth Australasian database conference on Database technologies 2003*, pages 217–226. Australian Computer Society, Inc., 2003.

[22] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: a database management system for semistructured data. *SIGMOD Rec.*, 26(3):54–66, 1997.

[23] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory*, pages 277–295. Springer-Verlag, 1999.

[24] Yannis Papakonstantinou and Victor Vianu. Incremental Validation of XML Documents. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 47–63. Springer-Verlag, 2002.

[25] Harald Schöning. Tamino - A DBMS designed for XML. In *Proceedings of the 17th International Conference on Data Engineering*, pages 149–154. IEEE Computer Society, 2001.

[26] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215. ACM Press, 2002.

[27] Zografoula Vagena, Mirella M. Moro, and Vassilis J. Tsotras. Twig query processing over graph-structured XML data. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 43–48. ACM Press, 2004.

[28] Xiaodong Wu, Mong Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *ICDE '04, Proceedings of the 20th International Conference on Data Engineering*, page 66. IEEE Computer Society, 2004.

[29] Liang Huai Yang, Mong-Li Lee, and Wynne Hsu. Efficient Mining of XML Query Patterns for Caching. In *VLDB*, pages 69–80, 2003.

[30] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Inter. Tech.*, 1(1):110–141, 2001.